If you have read other books on computers, you have probably noticed that we have made little mention of the binary number system. Most books begin with several chapters on this subject, something that we find very disturbing. Not that we object to the binary number system, you understand, it is the foundation of virtually all digital computers, including the ia7301. Our objection is to the placement of the subject at the beginning of the book where it bores the reader and discourages him from going on. Granted that any serious student of digital computers must learn the binary number system, but our approach has been to sneak up on it, so that when the time comes to discuss it fully the reader will already be somewhat familiar with binary numbers. That time is now.

We have already discussed the relationship between the decimal and hexadecimal number systems. We talked about how counting in the decimal system operates. Numerals 0—9 are used, and when that range is exceded, a new digit is introduced. Thus, the number after 9 is 10. Notice that when this step occurs, the right-most digit is reduced to the lowest value numeral, 0, and the digit to the left of it is incremented. This process continues as the count excedes the range of two digits, so that 99 is followed by 100. Hexadecimal operated in exactly the same fashion, except that there were 16 numerals, 0—F.

Computers are made up of electronic switching circuits that can distinguish between only two electrical states or conditions, ON and OFF. This makes the binary number system a much more natural choice for the computer, since the binary system has only two numerals, 0 and 1. For the purpose of this book we will make the OFF electrical state correspond to a binary 1 and an ON electrical state to a binary 0. There is nothing magical about this choice; it could have been the other way just as well, although in choosing as we did we follow the more common convention.

So why learn the binary number system? Because it's the system your computer uses, and you'll never really understand programming until you know it. There seems to be much confusion about this point, so we'll take a moment and elaborate. After all, haven't we been using hexadecimal to program the ia7301? You just thought you were! The ia7301 has a subroutine in its monitor operating in binary, as all programs do, which reads the keyboard and interprets the keys as binary numbers. When you press key A the computer reads it as 0000 1001, which is the binary equivalent of hexadecimal A. Thus, all the data, addresses and instructions going into the computer do so in binary even though it may appear to be hex. Likewise all the programs are executed in binary, results are stored in binary, addresses are sent in binary, and all I/O operations occur in binary. Then when the results are to be sent to the displays, another binary subroutine in the monitor, CONVERT already discussed, converts the binary numbers into seven-segment code equivalents that drive the LEDs in a hex format. Thus the ia7301 only appears to operate in hex. The same is true of any pocket calculator, except that here only 10 keys numbered 0–9 are present.

If the computer operates in binary, why use hexadecimal keyboards and displays? Well, for one thing binary numbers are long and clumsy for humans to use. How far do you think you would have gotten in this book if we had talked about address 1000 0011 1111 1111 instead of 83FFH? Hex numerals pack digits four times more efficiently than binary. Notwithstanding this fact, some microcomputer systems popular with hobbyists use a large array of binary toggle switches which make program loading a long and tedious affair.

Binary numbers are made up of 0's and 1's, called bits, short for BInary DigiTS. The ia7301 is organized as an 8-bit computer, which means that data and instructions are expressed in 8-bit words. Word lengths of 4, 8, 12, 16 and 32 bits are common in computers, although at this time most microcomputers use 8-bit words. This word length is so common, a special word has been invented for it, the *byte* (pronounced *bite*). Thus, the ia7301 memory stores one byte of data or instruction at each memory location.

We have listed below the binary, decimal and hexadecimal equivalents.

Binary	Decimal	Hexadecimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

Notice that a four-bit binary number is expressed with sometimes one, sometimes two decimal digits. This makes the binary to decimal conversion of long numbers difficult to do in your head. However, one hexadecimal digit can always represent a four bit binary number. Using the table, we can break down large binary numbers into four-bit groups (sometimes called *nibbles*) and then converted. Thus we can take 0111 1010, break it into two parts and convert the parts individually. The first part or nibble is 0111 and, looking this up in the table, we find the corresponding hex digit is 7. The second nibble is 1010 and we find from the table that this converts to A. Thus the hex equivalent of the byte 01111010 is 7A.

The ability to convert binary numbers into hexadecimal and back is one of the basic skills that every programmer must learn fairly early in the game. You should commit the hex to decimal chart to memory. Once the conversions for the first 16 numerals has been learned, larger numbers are easily converted by simply breaking them into nibbles and doing the conversions in four-bit increments.

Example: Convert 101100010111 to hexadecimal. Solution: Break the binary number into four-bit nibbles.

1011	0001	0111
B	1	7

Answer: B17H.

Example: Convert 10111100001010 to hexadecimal. This time the binary number does not break evenly into four bit nibbles. For that reason, leading zeroes are added to the most significant group to keep the nibbles intact.

0010	1111	0000	1010
2	F	0	A

Solution: 2F0AH.

Example: Convert 8F2AH into binary. This is the opposite of the conversions that we have just been doing. But the method is basically the same. Each of the hex numerals can be converted directly into its binary equivalent.

8	F	2	A
1000	1111	0010	1010

Answer: 1000111100101010.

Example: Convert 30FFH into binary.

3	0	F	F
0011	0000	1111	1111

Answer: 11000011111111.

Counting in Binary. One of the things that makes the binary number system so convenient for the computer is the smaller number of rules involved. For instance, while the decimal multiplication table has 100 entries for a ten by ten matrix, the binary system has only four entries. To appreciate how easy the system is to work with, we're going to go through the exercise of counting in this system.

When we count in the decimal system, we increment our count by repeatedly adding one to the current count. With ten numerals, this process continues until we try to increment the count at 9. At that point, we have run out of numerals, so we set the digit back to 0 and add one to the next digit. This gives us 10 and we continue until again we run out of numerals, so we increment the second digit again. The binary system is the same except with only two numerals, we run out a lot faster.

In the binary system, there are only two numerals, 0 and 1. If this system is really the same as the decimal system, we should be able to count by adding 1 in the first column, then when we run out of numerals, set the digit back to zero and increment the next digit. Except with only two numerals the process takes longer to describe than to do. Besides, we have a better idea. If we're such hot programmers, why not make the computer do the counting exercise for us. During the process we might even learn something about binary numbers.

The program that we will use is simplicity itself. We initialize the computer by loading the accumulator with 00H. This begins the counting process at zero. We then call a special subroutine we're going to load into the memory at 0200H. We'll go back to that subroutine in a second, but for now all we need to know is that the subroutine will convert the contents of the accumulator bits into 00H and 01H and store these numbers in the display registers so that they correctly represent the contents of the accumulator expressed in binary. If, for instance, the least significant bit of the accumulator is a zero, then the subroutine will load the display register no. 0 with 00H. If the next bit is a one, then the subroutine will load display register no. 1 with 01H. When the return from the subroutine occurs, we CALL CONVERT to convert these numbers into the correct seven segment codes and load those codes into the appropriate display buffers.

At this point we would have normally set up some sort of loop through KEYBOARD DISPLAY ONE-PASS to display the contents of the display buffers in the LEDs. Has it ever occured to you

that the computer somehow manages to display numbers on the LEDs and read the keyboard without you having to load that program in. After all, it goes through this very process whenever you turn the computer on. In fact, it does it as you load programs. The fact is that the computer has a version of the keyboard display program built right into the system monitor at location 80F2H. If you're interested, you can see the program at that location by examining the copy of the system monitor that appears in the service manual. We're not going to go into it here since it is not as powerful or complex as the keyboard display utility subroutine that we worked out in the last chapter. That utility subroutine was designed to make one scan of the displays and keyboard. If no key closure was found, the subroutine was exited through the normal RET mode. If a closure was found, it was debounced and an exit to a predetermined address occurred. The power of this subroutine was in the one pass dual exit that it offered, making it ideal for timing applications. But there are many applications that are not time dependent, and for them we can use the keyboard display subroutine that is built into the monitor. It has only a single exit mode, a standard RET, but is set up so the program loops within the subroutine continuously displaying the contents of the display buffers until a key closure occurs. The closure is then debounced and an exit through the RET is executed.

Conceptually this is a much simpler subroutine. When it is CALLed, the program will remain in that subroutine driving the display until a key is pressed. The subroutine is immediately exited with the image of the key in the accumulator. Since there is only one exit mode, there is no need to follow the CALL with the jump destination as we did in the case of our utility subroutine. You'll find the starting address of this monitor version of the keyboard display subroutine listed at the bottom of the hex card along with some of the other useful monitor routines.

When a key closure occurs, we have to decide whether or not to use it. Our purpose in waiting for the key closure is to cue the computer when to perform the count. For that we don't want or need every key on the keyboard to cause the computer to count. Instead, we'll use just the

NEXT key. If you check back to the table we asked you to fill out in the last chapter, you'll find that the NEXT key causes the computer to return from the subroutine with 12H in the accumulator. We can test for that key by doing a CPI 12H and then a JNZ back to the beginning of the program. If the key closure turns out to be the NEXT key, then we increment the accumulator and go back to the beginning. The program thus operates by displaying all zeroes in the displays, then counting up in binary every time the NEXT key is pressed. The flow diagram for the program is illustrated in Fig. 9-1. Load the following program into your computer.

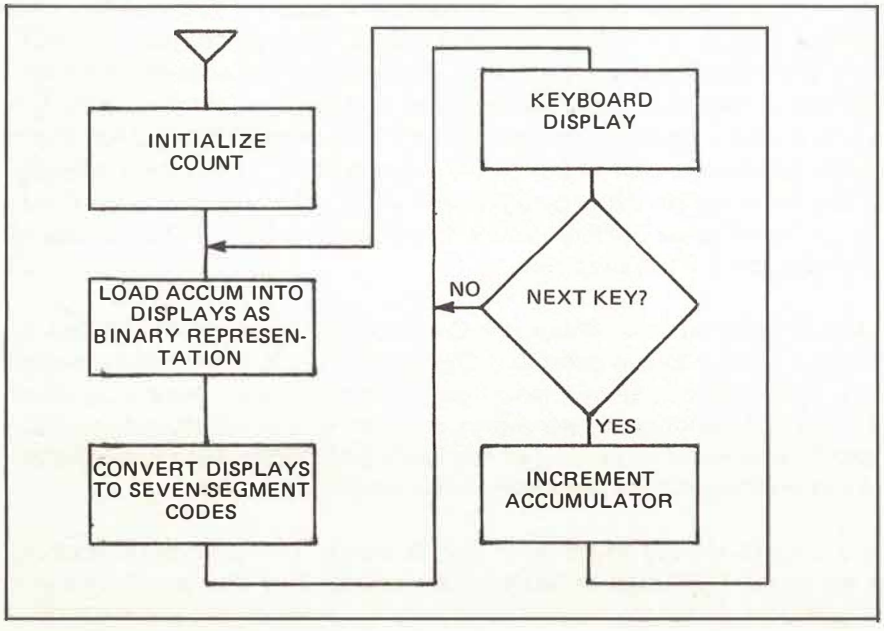


Fig. 9-1 Overall flow diagram for binary counting program.

```
0100    3E      MVI     A, 00H          ;Initialize the computer to the
0101    00                              ;first count, zero.
0102    CD      CALL DISP BINARY (0200H)  ;This subroutine converts the
0103    00                              ;contents of the accumulator into
0104    02                              ;a binary representation in the
0105    F5      PUSH PSW                ;display buffers. Save accum.
0106    CD      CALL CONVERT (8132H)    ;Convert contents of display
0107    32                              ;registers into seven segment codes
0108    81                              ;in the display buffers.
0109    CD      CALL KEYBD-DSPLY (80F2H)  ;Display contents of buffers and
010A    F2                              ;wait for key closure.
010B    80                              ;
010C    FE      CPI 12H (NEXT)          ;Check to see if closure is from
010D    12                              ;NEXT key. If not, go
010E    C2      JNZ     0109H           ;back to keyboard display.
010F    09                              ;
0110    01                              ;
0111    F1      POP     PSW             ;Restore count to accum. Then
0112    3C      INR     A               ;increment it and loop back to
0113    C3      JMP     0102H           ;beginning.
0114    02                              ;
0115    01                              ;
```

This is pretty straight forward and follows our earlier description very closely. With the exception of the mystery subroutine that is still to be discussed, you should be able to follow the program from the comments that accompany it. You'll notice that the current count in the accumulator is saved in the stack so that a key closure in the keyboard display subroutine will not erase that count. As soon as a closure is found, the count is POPed back off the stack into the accumulator and incremented with a INR A instruction. In the event that a key other than

NEXT is pressed, we loop back to the keyboard display subroutine and wait for another key closure.

Actually without the subroutine at 0200H, there is nothing to tell us that this program is displaying its count in binary. The counting process itself is going on in the usual fashion, and without the subroutine, the count could have been displayed in hex very easily. The subroutine is listed below and illustrated in a flow diagram in Fig. 9-2. Enter the subroutine into the computer.

DISPLAY BINARY

```
0200  21   LXI   H, 000FH   ;Point H/L to the left-most
0201  0F                    ;display register.
0202  00                    ;
0203  16   MVI   D, 08H     ;Set up D as a loop count for
0204  08                    ;eight passes.
0205  07   RLC               ;Rotate accum one bit position to
0206  DA   JC    0211H      ;the left.  If this rotates a 1
0207  11                    ;into the carry, jump to 0211H.
0208  02                    ;If not, fall through and load
0209  36   MVI   M, 00H     ;00H into the display register.
020A  00                    ;
020B  15   DCR   D          ;Decrement loop counter.
020C  2B   DCX   H          ;Point H/L to next display register.
020D  C8   RZ               ;If loop counter zero, return to
020E  C3   JMP   0205H      ;CALLing program.  If not, loop
020F  05                    ;back again for another pass.
0210  02                    ;
0211  36   MVI   M, 01H     ;Load 01H into the display register.
0212  01                    ;
```

```
0213    C3    JMP    020BH        ;Go back to the decrement and
0214    0B                        ;test portion of the program to
0215    02                        ;see if an exit should occur.
```

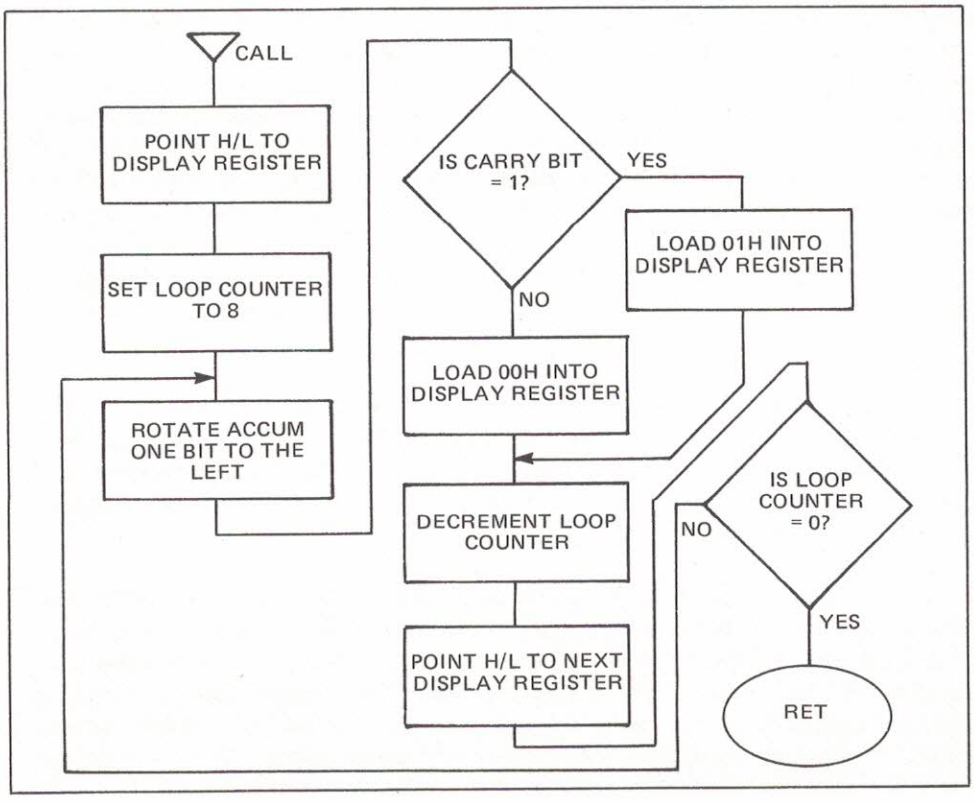


Fig. 9-2 Flow diagram of sub-routine to display contents of accumulator in binary.

That certainly doesn't look very imposing, does it? The subroutine operates by either loading a 00H or a 01H into a display register. By pointing H/L to the left-most display register and setting up D as a loop counter, the loading process can be done easily. The question really comes down to, whether we load 00H or 01H into any given register. Obviously the answer to that question is tied up in the binary contents of the accumulator. You'll remember at the point that this subroutine is CALLed, the current count is contained in the accumulator. Do we have to perform some sort of exotic hexadecimal to binary conversion on those contents? No, as we stated very strongly in the beginning of this chapter, the computer operates in binary 100% of the time; it is only at the input and output that we convert through one of our monitor programs into the more easily digestible hex system. The accumulator, like the other working registers, is an 8-bit register and the contents are stored there in the forms of 0's and 1's (actually electrically lows and highs to which we have assigned the 0 and 1 convention). All our subroutine must do is determine whether a given bit is 0 or 1 and then load the appropriate 00H or 01H into the corresponding display register.

There are many ways to test whether a given bit is 0 or 1. The method that we use here is the best when multiple bits must be tested, one at a time. If only one bit need be tested, the ANI followed by a JZ test is generally shorter. Our method involves rotating the accumulator and then testing the carry bit to see if we have rotated in a 0 or a 1. Let's review the rotate instructions. See Fig. 9-3.

All of the four rotate accumulator instructions cause the carry flag to be loaded with the contents of one of the extreme end bits of the accumulator; which end is determined by the direction of rotation. In the RLC and RRC instructions, the bit that is being shifted all the way around the accumulator is duplicated into the carry flag position. The rotate operation itself is an 8-bit rotation; the former contents of the carry flag are lost. In the RAL and RAR instructions, the accumulator and the flag are combined into a nine bit accumulator. In this case, the

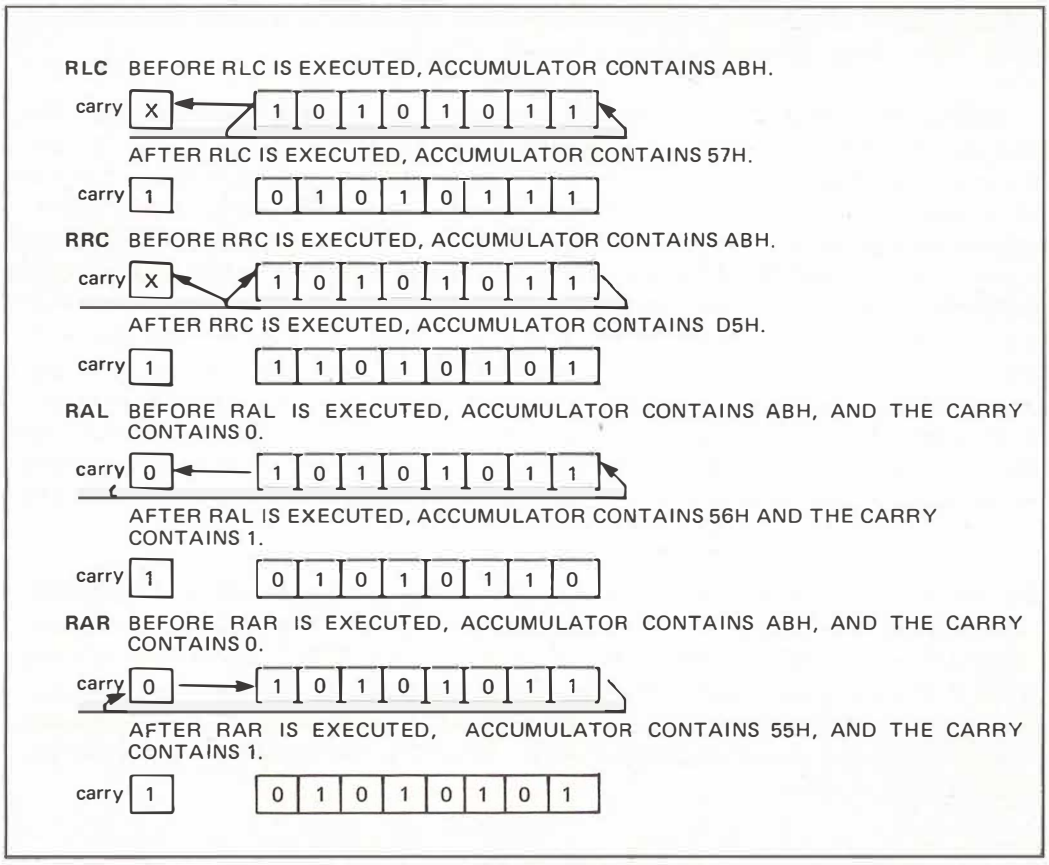


Fig. 9-3 The four accumulator rotate instructions summarized.

former contents of the flag bit are shifted into the accumulator and in turn, is loaded with one of the accumulator bits. The rotate operation is now a nine bit rotation.

Since the rotate accumulator instructions cause the carry flag to be loaded with one of the extreme bits of the accumulator, we can use it to examine the accumulator, bit by bit. In conjunction with the rotate accumulator, we will need to be able to test the carry flag to see if it is a 0 or a 1. That is easily done with the JC, Jump if Carry, and JNC Jump it No Carry, In the program subroutine that we just loaded, either the RLC or the RRC instructions will work. These are chosen over the RAL and RAR because the latter involve a nine-bit rotate and we wish to preserve the contents of the accumulator after eight load display operations. Since the RLC and RRC involve an eight bit rotation, they present a slightly simpler solution. After the RLC has been executed, the carry flag has now been loaded with what was formerly the most significant bit of the accumulator, a JC, Jump if Carry, instruction tests this flag. If it is 1, meaning a carry is present, we jump off to a point further along in the program where a 01H is loaded into the display register. If the JC test fails, there must have been a 0 in the most significant bit of the accumulator. We fall through the JC test and load a 00H into the display register with a MVI M instruction.

Now to decrement the loop counter and move on to the next display register. DCR D accomplishes the former and DCX H, the latter. Is the loop counter now zero? If it is, we should return to the CALLing program with a RZ, Return if Zero. Notice that the DCR D instruction sets the zero flag, but the DCX H instruction has no effect on any of the flags. So the zero flag from the DCR D is still intact after DCX H and the RZ test will be valid. This business of which flags are set by which instructions must be monitored very closely. All of the flag operations are noted on the hex card to facilitate this process.

Let's try it. Press CLR and EXC. You should see a row of zeroes on the displays. Try pressing any of the keys except NEXT. They should have no effect, since we specifically wrote our program to ignore them. Then press NEXT. The displays should increment, just as they would have done in the decimal system.

Enter:	See Displayed:
CLR EXC	0 0 0 0 0 0 0 0
NXT	0 0 0 0 0 0 0 1

By the rules for counting that we worked out earlier, incrementing the count again makes us run out of numerals in the first digit column, after all, there is no such thing as a 2 in binary. So we set the first column back to 0 and increment the next column. Since it's easier to do than describe, press NEXT again.

NXT	0 0 0 0 0 0 1 0

Each time we press NEXT, the binary count will be incremented. Doing it now, we increment the first digit, and since that does not cause us to run out of numerals, the next digit is unchanged.

NXT	0 0 0 0 0 0 1 1

Incrementing the count again, brings us to a situation similar to that of trying to count past 99. When we increment the first digit, we find it must be set back to 0 and the next digit incremented instead. But it too is already at 1 and must be set back to 0. To count past this point we must increment the third digit.

NXT 00000100

We could continue with this process indefinately, but you're doubtless getting the idea already. If not, play with this program for a while. You should be able to predict the next count before pressing the key before leaving this program.

Binary Arithmetic and Logical Operations. Now that we understand how to count and convert binary numbers, let's explore the rules by which arithmetic and logical operations are done on these numbers. So far in this course, we have briefly discussed AND, OR and EXCLUSIVE OR logical operations, but without a firm understanding of the binary systems, these discussions were of necessity abbreviated. While we have done arithmetic operations, it was possible to discuss these in the context of hexadecimal numbers, so that the lack of binary understanding did not stop us from using these instructions. We are about to fill this binary void.

In the same way that we constructed a program for counting in binary, we are about to write and execute a program that will act as a test bed for binary operations, a sort of binary laboratory, if you will. That program will use the subroutine for displaying the contents of the accumulator in binary form in the displays, so don't turn the computer off. While the entire program is rather involved, it is invaluable as a way to quickly check the results of arithmetic or logical operations within the computer. For that reason, we suggest that you commit the program to tape, so you can load it in and use it whenever you get stuck on some sort of binary operation.

As part of the program we need to develop the counterpart to the subroutine that loaded the accumulator in a binary representation into the displays. This second subroutine, located at 0216H, acts to take the binary number represented in the displays and load it into the accumulator. The flow diagram for the subroutine is illustrated in Fig. 9-4. Load the listing below into the computer.

BINARY DISPLAYS TO ACCUMULATOR

```
0216    E5      PUSH    H           ;Save the H/L/D/E registers on
0217    D5      PUSH    D           ;the stack.
0218    21      LXI     H, 000FH    ;Point H/L to left-most display
0219    0F                          ;register.
021A    00                          ;
021B    16      MVI     D, F8H      ;Set up D as a loop counter.
021C    F8                          ;
021D    35      DCR     M           ;Decrement contents of display
021E    CA      JZ      0226H       ;register by 1.  If it now contains
021F    26                          ;0, it must have been 01 before.
0220    02                          ;If so, go to 0226H and set carry.
0221    37      STC                 ;If not, load a 0 into carry and
0222    3F      CMC                 ;rotate into the accumulator.
0223    C3      JMP     0227H       ;
0224    27                          ;
0225    02                          ;
0226    37      STC                 ;Set carry to 1 and rotate into
0227    17      RAL                 ;the accumulator.  Point H/L to
0228    2B      DCX     H           ;next display register.  Increment
0229    14      INR     D           ;loop counter and test to see if
022A    C2      JNZ     021DH       ;zero.  If not, go back and loop
022B    1D                          ;through again.  If zero, exit.
022C    02                          ;
022D    D1      POP     D           ;
022E    E1      POP     H           ;
022F    C9      RET                 ;
```

The subroutine has some interesting features. It begins by storing the registers used in the subroutine in the stack memory. H/L are then pointed to the left-most display register, and the D register is set up as a loop counter for keeping track of the number of display registers that have been

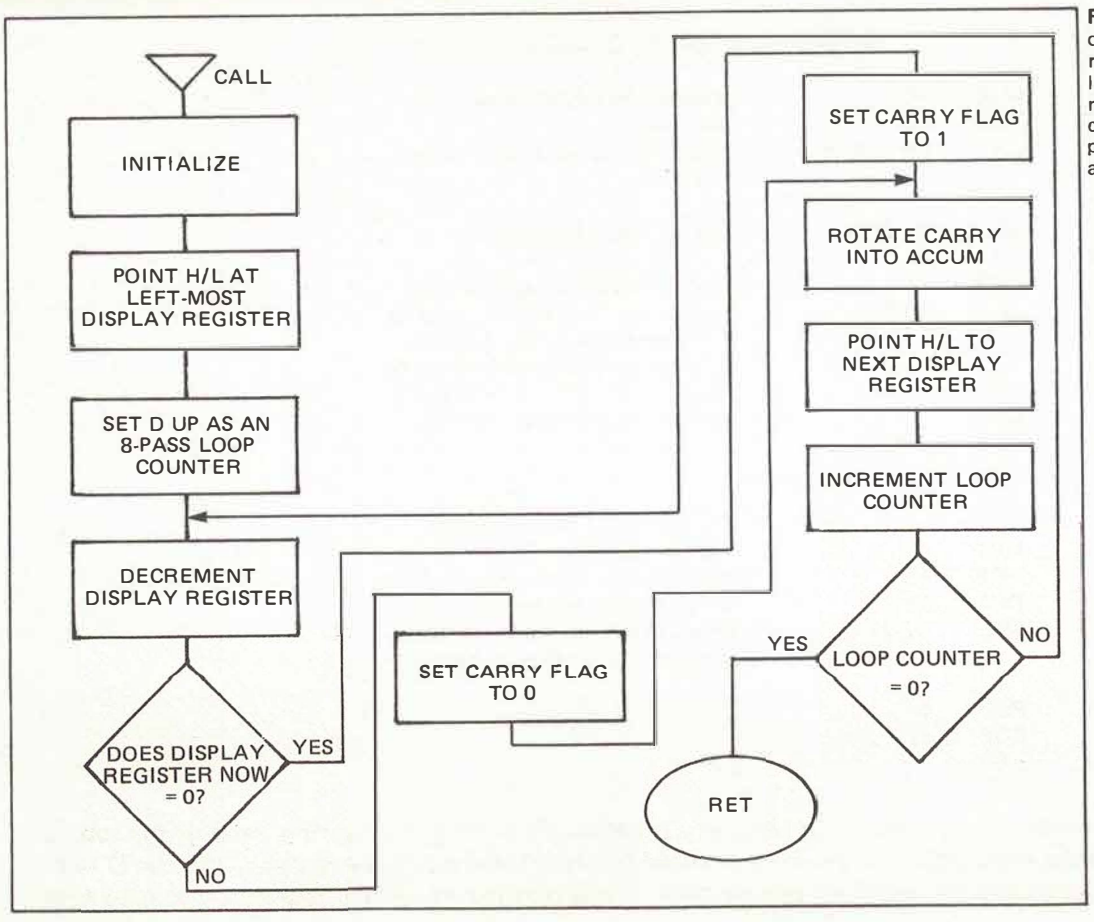


Fig. 9-4 Flow diagram of subroutine used to load binary representation contained in display registers into accumulator.

examined. Just for fun, we treated the loop counter as an ascending counter rather that the descending that we have been using up until now. In this type, the counter is set at F8H and is incremented on each pass. After the eighth such pass through the loop, the counter will reach 00H, after going through FFH, and the JNZ test will fail and the program will come out of the loop. The INR D instruction used for this loop is similar to the other INR R instructions already used.

Inside of the loop is a test designed to determine if the display register contains 00H or 01H. A DCR M instruction is used. If the register contained 01H prior to the test, decrementing it will cause it to contain 00H and the zero flag will be set. On the other hand, if it already contained 00H, the decrement operation will cause it to contain FFH and the zero flag will not be set. Assume that the register originally contained 01H so that the DCR M and JZ test will cause the jump. The jump is made to a STC, Set Carry, instruction which causes the carry flag to be set to 1. This is followed by a RAL instruction which rotates the carry flag, now 1, into the right-most bit position of the accumulator. Since there will be a rotate left operation in each of the eight passes through the loop, at the end of the subroutine the 1 that we just loaded into the right end of the accumulator will have been shifted to the left so that it ends up in the left-most bit position of the accumulator. Thus we have successfully converted the 01H contents of the left-most display register into a 1 in the left-most bit position of the accumulator. If the contents of the register had been a 0 instead, the JZ test would have failed and the program execution would have fallen through to another STC followed by a CMC, Complement Carry, instruction. Notice that we do not have an instruction that sets the carry flag to a 0, but we have the next best thing, a CMC. This instruction changes or complements the carry flag, so that a STC followed by a CMC will have the end effect of setting the carry to 0. In our program, the STC, CMC sequence is followed by a jump to the RAL instruction of the other branch since, from this point on, the branches are identical.

Load the remainder of the program into the computer.

BINARY LABORATORY

```
0100  3E  MVI   A, FFH          ;Load FFH into the discrete LEDs
0101  FF                       ;to blank them.  Also load this
0102  32  STA   6000H           ;value into a memory location, 001DH,
0103  00                       ;which serves as a mirror of the
0104  60                       ;data in the LEDs.  This is so
0105  32  STA   001DH           ;the computer can determine the
0106  1D                       ;state of the LEDs since it cannot
0107  00                       ;read them directly.
0108  CD  CALL BLANK (80CCH)   ;Blank the displays.
0109  CC                       ;
010A  80                       ;
010B  CD  CALL CONVERT (8132H) ;Convert the contents of the display
010C  32                       ;registers to 7-segment
010D  81                       ;code in the display buffers.
010E  CD  CALL KB-DSPLY (80F2H);Drive displays and wait for a
010F  F2                       ;key closure.
0110  80                       ;
0111  FE  CPI   NEXT (12H)     ;Is the closure from the NEXT key?
0112  12                       ;If so, go to 013AH.
0113  CA  JZ    013AH          ;This key is used to cue the computer
0114  3A                       ;that the user is satisfied with
0115  01                       ;the number appearing on the displays
0116  FE  CPI   02H            ;Is the key either 0 or 1?  If so
0117  02                       ;go to 0121H to process it.
0118  DA  JC    0121H          ;
0119  21                       ;
011A  01                       ;
```

```
011B  CA    JZ      012CH       ;Process the no. 2 key.
011C  2C                        ;
011D  01                        ;
011E  C3    JMP     010BH       ;This is the default jump.  If
011F  0B                        ;none of the other conditions
0120  01                        ;have caused a jump, this is an
0121  21    LXI     H, 000FH    ;invalid key and we should ignore
0122  0F                        ;it.  Go back to the keyboard.
0123  00                        ;Point H/L to the left-most display.
0124  0E    MVI     C, 08H      ;By pointing C to 08, we are set
0125  08                        ;up for SUB1C, the shift and enter
0126  CD    CALL SUB1C (807CH)  ;subroutine.  This takes the contents
0127  7C                        ;of the accum and shifts it into the
0128  80                        ;right-most display, shifting each
0129  C3    JMP     010BH       ;display one digit to the left
012A  0B                        ;Go back to the keyboard for the
012B  01                        ;next digit.

012C  3A    LDA     001DH       ;This is the entrance for the no. 2 key
012D  1D                        ;process program.  The object is to
012E  00                        ;complement the carry flag.  Load
012F  EE    XRI     08H         ;the accum with the image of the
0130  08                        ;discrete LEDs.  Complement the bit
0131  32    STA     001DH       ;corresponding to the carry LED and
0132  1D                        ;send the result back to the memory
0133  00                        ;and to the discrete LEDs.
0134  32    STA     6000H       ;
0135  00                        ;Then go back to the keyboard.
0136  60                        ;
0137  C3    JMP     010BH       ;
0138  0B                        ;
0139  01                        ;
```

```
013A  CD     CALL DIS TO ACC (0216H)   ;This point has been reached because
013B  16                               ;the NEXT key was pressed. The sub
013C  02                               ;causes the displays to be loaded into
013D  47     MOV    B, A               ;the accumulator. We then save it
013E  CD     CALL BLANK (80CCH)        ;in the B register. Blank the
013F  CC                               ;displays.
0140  80                               ;
0141  CD     CALL CONVERT (8132H)      ;Convert the contents of the display
0142  32                               ;registers into 7-segment code in
0143  81                               ;the display buffers.
0144  CD     CALL KEY-DSPLY (80F2H)    ;Display contents of buffers and
0145  F2                               ;wait for key closure.
0146  80                               ;
0147  FE     CPI    02H                ;Check to see if the key was either
0148  02                               ;a 0 or a 1. If so, go to 015BH.
0149  DA     JC     015BH              ;
014A  5B                               ;
014B  01                               ;
014C  FE     CPI    10H                ;Check to see if the key closure
014D  10                               ;is one of the mode keys. If so
014E  D2     JNC    0144H              ;go back to the keyboard. If
014F  44                               ;not, save the accumulator in
0150  01                               ;the D register and check to be
0151  57     MOV    D, A               ;sure the key is greater than
0152  3E     MVI    A, 08H             ;eight. If not, go back to the
0153  08                               ;keyboard.
0154  BA     CMP    D                  ;
0155  D2     JNC    0144H              ;
0156  44                               ;
0157  01                               ;
0158  C3     JMP    0166H              ;Go to the arithmetic and logical
0159  66                               ;operation key process program.
015A  01                               ;
```

```
015B  21    LXI   H, 000FH      ;Process 1 and 0 keys. Point H/L
015C  0F                        ;to left-most display register.
015D  00                        ;
015E  0E    MVI   C, 08H        ;Load C with 08H to set up computer
015F  08                        ;for SUB1C.
0160  CD    CALL SUB1C (807CH)  ;Shift the contents of the accum
0161  7C                        ;into the right hand display and
0162  80                        ;shift each of the displays one
0163  C3    JMP   0141H         ;digit to the left. Then jump
0164  41                        ;back for the next key closure.
0165  01                        ;
0166  CD    CALL DIS TO ACC (0216H) ;This point has been reached because
0167  16                        ;the user pressed keys 9-F. Convert
0168  02                        ;the displays to a number in the
0169  4F    MOV   C, A          ;accum and then save in the C reg.
016A  7A    MOV   A, D          ;Fetch the key image from D to accum.
016B  07    RLC                 ;Multiply this number by four.
016C  07    RLC                 ;
016D  21    LXI   H, 015FH      ;Point H/L at the starting address.
016E  5F                        ;of the jump table.
016F  01                        ;Add L to the accum and then move
0170  85    ADD   L             ;the result back to L so that H/L
0171  6F    MOV   L, A          ;now contain the correct address in
0172  3A    LDA   001DH         ;the jump table. Load the image
0173  1D                        ;of the discrete LEDs into the accum
0174  00                        ;and mask out all but the carry bit.
0175  E6    ANI   08H           ;Test to see if this bit is zero
0176  08                        ;(meaning there is a carry) or not.
0177  B7    ORA   A             ;If there is a carry bit = 0, jump
0178  CA    JZ    0180H         ;to 0180H to set the real carry flag
0179  80                        ;to 1. If the carry bit = 1, fall
017A  01                        ;through to 017BH to set the real
```

```
017B  37    STC                 ;carry flag to 0.
017C  3F    CMC                 ;
017D  C3    JMP     0181H       ;Jump to exit.
017E  81
017F  01                        ;
0180  37    STC                 ;Set carry. Then load first operand
0181  78    MOV     A, B        ;into accumulator and jump to
0182  E9    PCHL                ;address in H/L pair.
0183  A1    ANA     C           ;This is the first entry in the
0184  C3    JMP     019FH       ;table of instructions. Add the
0185  9F                        ;operand in the C register to the
0186  01                        ;operand in the accumulator and
0187  B1    ORA     C           ;jump to the exit point, 019FH.
0188  C3    JMP     019FH       ;This is repeated for the other
0189  9F                        ;instructions in the repetoire.
018A  01                        ;Thus key no. 9 performs ANA. Key no. A
018B  A9    XRA     C           ;performs ORA, key no. B performs
018C  C3    JMP     019FH       ;XRA, key no. C performs ADD, key no. D
018D  9F                        ;performs ADC, key no. E performs SUB,
018E  01                        ;and key no. F performs SBB. In each
018F  81    ADD     C           ;case a jump to 019FH follows the
0190  C3    JMP     019FH       ;logical or arithmetic operation.
0191  9F                        ;
0192  01                        ;
0193  89    ADC     C           ;
0194  C3    JMP     019FH       ;
0195  9F                        ;
0196  01                        ;
0197  91    SUB     C           ;
0198  C3    JMP     019FH       ;
0199  9F                        ;
019A  01                        ;
019B  99    SBB     C
```

```
019C    C3      JMP     019FH           ;
019D    9F                              ;
019E    01                              ;
019F    4F      MOV     C, A            ;Save the result in the C register.
01A0    3F      CMC                     ;We will now light the LEDs
01A1    17      RAL                     ;with the data in the flags. Comple-
01A2    EA      JPE     01A9H           ;ment the carry and shift into the
01A3    A9                              ;accumulator. If the parity is
01A4    01                              ;even, set the carry to zero and
01A5    37      STC                     ;shift into the accum. If not, shift
01A6    C3      JMP     01ABH           ;in a one. The LED for the sign
01A7    AB                              ;flag is set at 01ACH.
01A8    01                              ;
01A9    37      STC                     ;
01AA    3F      CMC                     ;
01AB    17      RAL                     ;
01AC    FA      JM      01B3H           ;If the result is minus, shift in
01AD    B3                              ;a zero. If positive, shift in a
01AE    01                              ;one.
01AF    37      STC                     ;
01B0    C3      JMP     01B5H           ;
01B1    B5                              ;
01B2    01                              ;
01B3    37      STC                     ;
01B4    3F      CMC                     ;
01B5    17      RAL                     ;
01B6    17      RAL                     ;Rotate once more to bring the 3 LED
01B7    F6      ORI     F1H             ;bits into the correct place in the
01B8    F1                              ;accum. Load the other 5 bits with
01B9    32      STA     6000H           ;ones. Send the data to the LEDs
01BA    00                              ;and the memory location 001DH
01BB    60                              ;reserved for the image of the LEDs.
```

```
01BC    32      STA     001DH           ;
01BD    1D                              ;
01BE    00                              ;
01BF    79      MOV     A, C            ;Restore the result stored in the
01C0    CD      CALL 0200H              ;C register to the accumulator.
01C1    00                              ;Display the contents of the
01C2    02                              ;accumulator in binary form in the
01C3    C3      JMP     010BH           ;LEDs. Then go back to 010BH and
01C4    0B                              ;get a new operand.
01C5    01                              ;
```

Because of the complexity of this program, we'll go through it in some detail. The flow diagram of the entire program is illustrated in Fig. 9-5, and you should refer frequently to this diagram while following the discussion. The purpose of the program is to let the user load in two binary numbers and then operate on them with any of six logical or arithmetic instructions. The results will be displayed as will the effect on the various flags. Because the program allows the user to set the carry flag himself, before performing the specified operation, we require a memory location for storing the status of the flags. The contents of this location are then used to drive the three discrete LEDs to form a display of three of the flags, carry, sign and parity.

Because the three discrete LEDs will be used to display the status of the flags, we will blank them as part of the initialization of the program. This is done by sending FFH to the displays at 6000H. This same data is then sent to 001DH, the memory location reserved for the image of the LEDs. From our discussions of the binary number system, we now know that FFH is the hex representation of an 8-bit binary number made up entirely of 1's. It takes a 0 in the correct bit position to light one of the LEDs, so FFH must blank all three LEDs.

The next step is to blank the display LEDs so the previous contents of the display registers will be erased. As we have seen in earlier chapters, the BLANK subroutine at 80CCH can be used to accomplish this automatically. We then enter the display loop portion of the program. This is

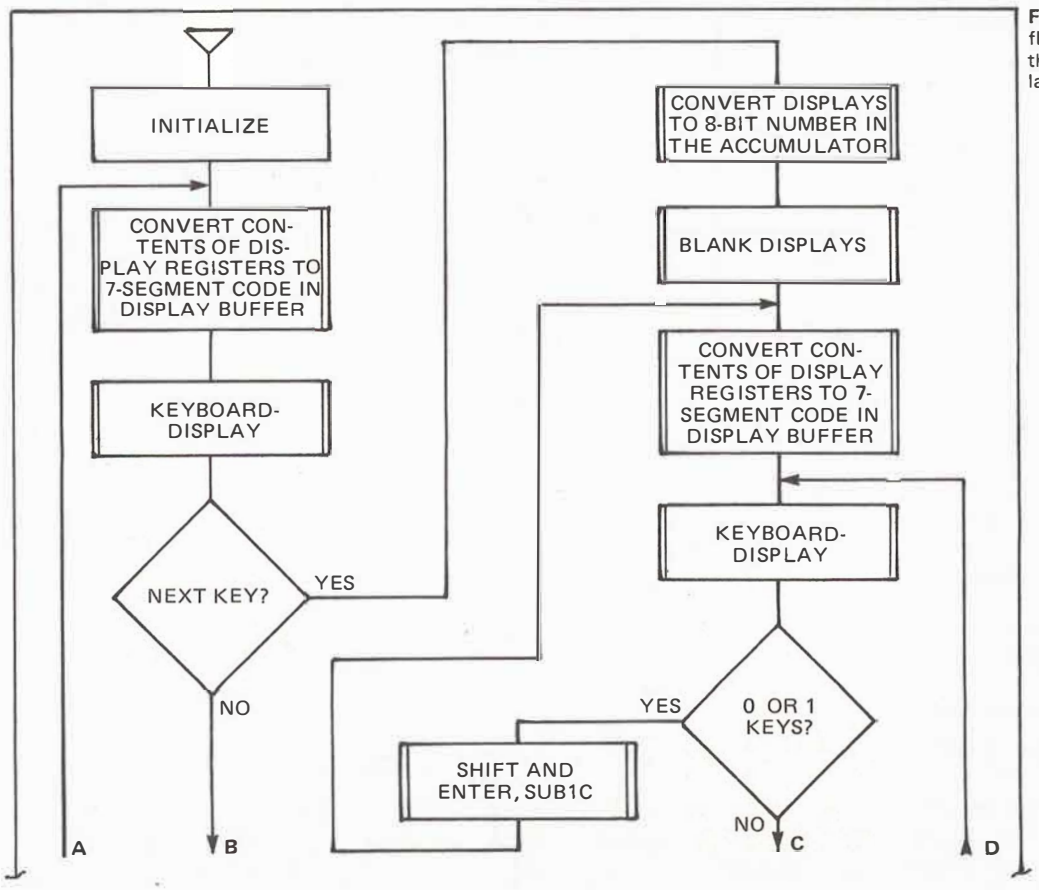
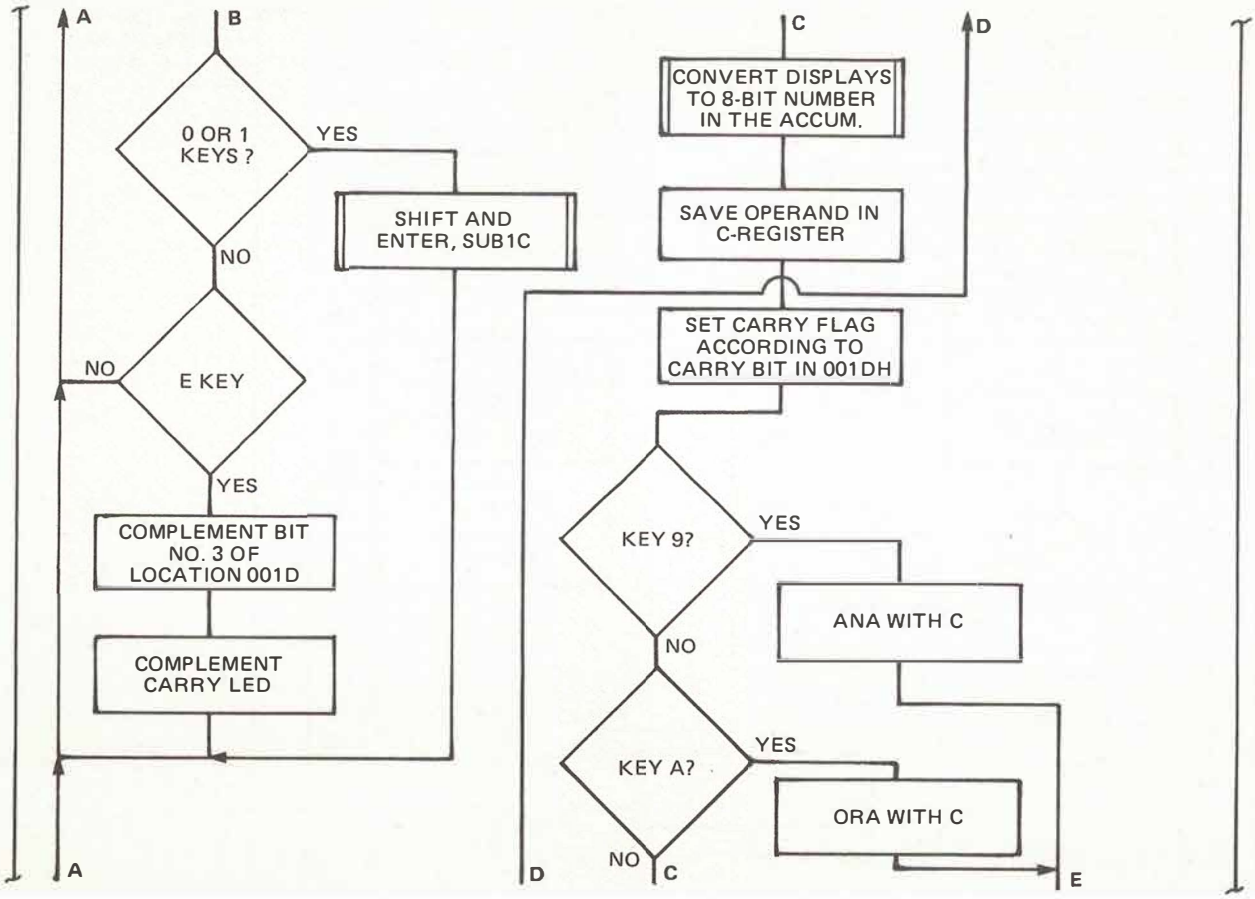
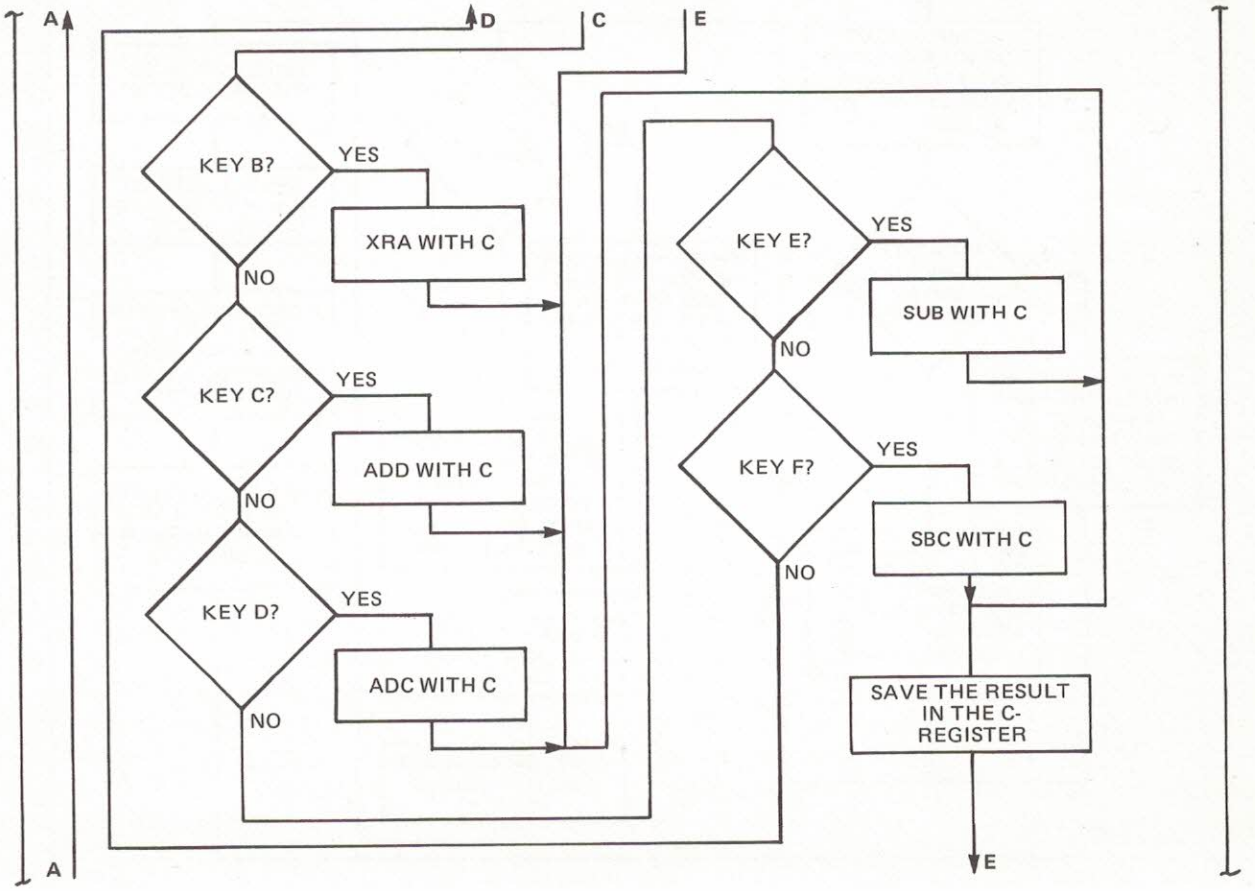
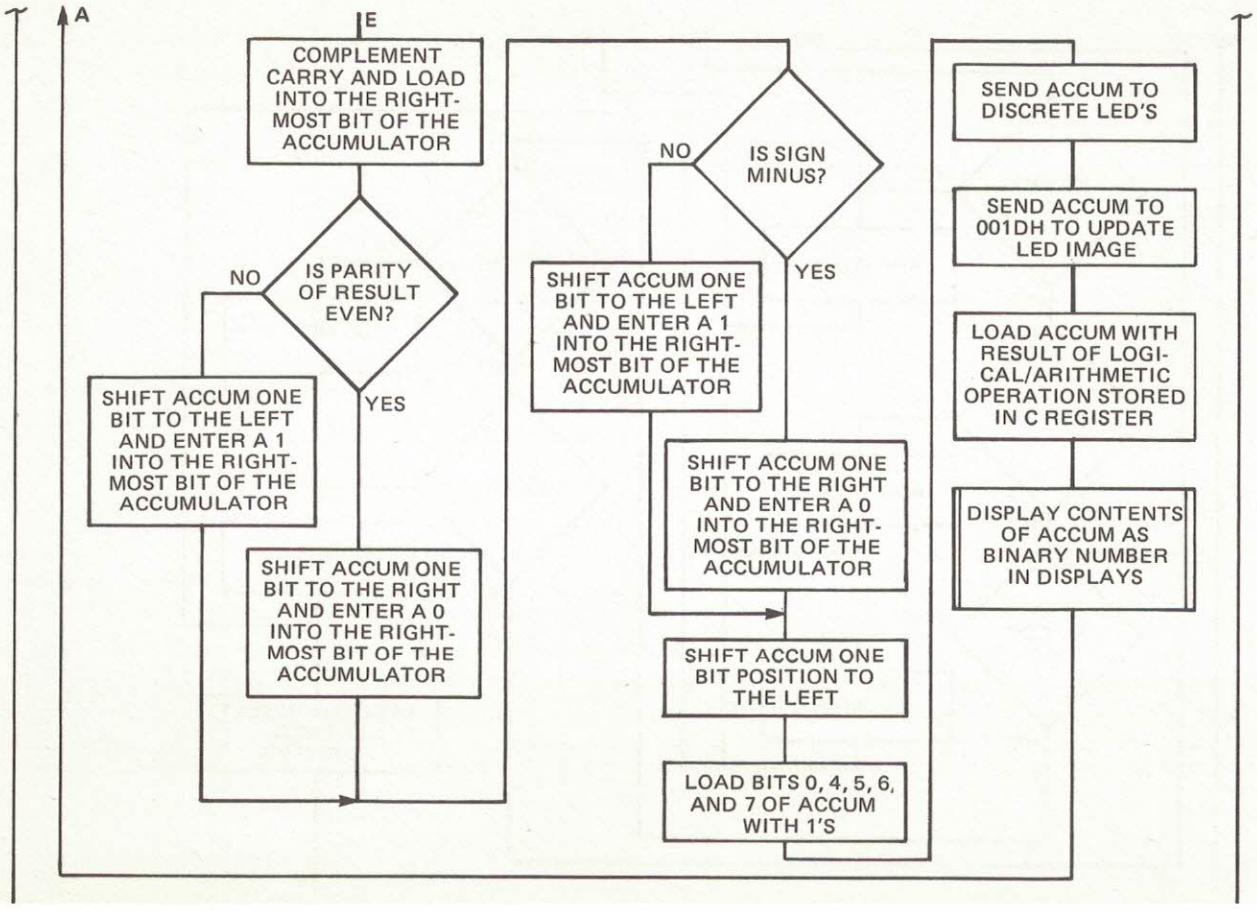


Fig. 9-5 Overall flow diagram for the binary laboratory.


A
B
0 OR 1 KEYS ?
YES
NO
SHIFT AND ENTER, SUB1C
NO
E KEY
YES
COMPLEMENT BIT NO. 3 OF LOCATION 001D
COMPLEMENT CARRY LED
A
C
CONVERT DISPLAYS TO 8-BIT NUMBER IN THE ACCUM.
SAVE OPERAND IN C-REGISTER
D
SET CARRY FLAG ACCORDING TO CARRY BIT IN 001DH
KEY 9?
YES
ANA WITH C
NO
KEY A?
YES
ORA WITH C
NO
C
D
E


A
D
C
E
KEY B?
YES
XRA WITH C
NO
KEY C?
YES
ADD WITH C
NO
KEY D?
YES
ADC WITH C
NO
KEY E?
YES
SUB WITH C
NO
KEY F?
YES
SBC WITH C
NO
SAVE THE RESULT IN THE C-REGISTER
E
A


A
E
COMPLEMENT CARRY AND LOAD INTO THE RIGHT-MOST BIT OF THE ACCUMULATOR
IS PARITY OF RESULT EVEN?
NO
YES
SHIFT ACCUM ONE BIT TO THE LEFT AND ENTER A 1 INTO THE RIGHT-MOST BIT OF THE ACCUMULATOR
SHIFT ACCUM ONE BIT TO THE RIGHT AND ENTER A 0 INTO THE RIGHT-MOST BIT OF THE ACCUMULATOR
IS SIGN MINUS?
NO
YES
SHIFT ACCUM ONE BIT TO THE LEFT AND ENTER A 1 INTO THE RIGHT-MOST BIT OF THE ACCUMULATOR
SHIFT ACCUM ONE BIT TO THE RIGHT AND ENTER A 0 INTO THE RIGHT-MOST BIT OF THE ACCUMULATOR
SHIFT ACCUM ONE BIT POSITION TO THE LEFT
LOAD BITS 0, 4, 5, 6, AND 7 OF ACCUM WITH 1'S
SEND ACCUM TO DISCRETE LED'S
SEND ACCUM TO 001DH TO UPDATE LED IMAGE
LOAD ACCUM WITH RESULT OF LOGICAL/ARITHMETIC OPERATION STORED IN C REGISTER
DISPLAY CONTENTS OF ACCUM AS BINARY NUMBER IN DISPLAYS

begun with a CALL to CONVERT to convert the contents of the display registers into their seven segment equivalents in the display buffers. Of course, at this point the registers contain the code for blanks from the BLANK subroutine, but the next time through the loop that will no longer be true. The actual display is executed by a CALL to KEYBOARD DISPLAY at 80F2H. This is the same subroutine that we used in the binary counting program earlier in this chapter. It caused the contents of the display buffers to appear in the appropriate LEDs, and remains in that mode until a valid key closure occurs.

At this point in the program only keys 0 , 1 , 2 or NEXT are considered valid closures. The 0 and 1 are used to input the binary data into the displays. Use of these keys allows the user to experiment with any binary number he chooses. The 2 key is used to complement the carry flag. Since the status of this flag is constantly reflected in the top LED in the lower left corner of the computer, the user can change the state of the carry flag by pressing the 2 key and watching the result on the LED. When the user is satisfied with the state of the carry and the binary number that he has loaded into the displays, he presses NEXT which cues the computer to process the data.

When a key closure occurs, the computer exits the KEYBOARD DISPLAY subroutine with the image of the key in the accumulator. It is easy to test the key to see if the NEXT key was pressed. This is done with a CPI 12H followed by a JZ 013AH. Since the numeric value associated with the NEXT key is 12H, performing a CPI 12H will result in the zero flag being set if the comparison is successful. The JZ 013AH then sends program execution off to 013AH to process the data in the displays. The NEXT key cues the computer when the number in the displays is correct, and obviously that is not yet the case. Thus the test will fail and the program will fall through to 0116H where we test for the 0 , 1 , and 2 keys. By being clever in our choice of CPI and JMP instructions, we can test for all three at once.

We know that when a CMP or CPI instruction is executed, if the two quantities are the same, the zero flag will be set to 0. A JZ or JNZ instruction can then be used to test the outcome of the comparison. As it turns out, the carry flag is also set by that same comparison. If the immediate data, in the case of CPI, or the data in the register, in the case of CMP, is greater than the data in the accumulator, the carry flag will be set to 1. In our present program, we execute a CPI 02H instruction. If the key closure was either the 0 key or the 1 key, the immediate data, 02H, will be greater than the data in the accumulator and the carry flag will be set to 1. A JC 0121H instruction tests this and jumps to 0121H to process the 0 or the 1 keys if that turns out to be the situation. If this test fails, there remains the possibility that the 2 key was pressed. The CPI 02H set the zero flag according to the comparison with the immediate data 02H, and that flag is still set. If the 2 key was pressed, then the JZ 012CH test will cause a jump to 012CH to process the carry bit in the 012CH register. Finally, the user may have pressed one of the other keys which has to be treated as an error. Rather than leave the situation to a chance, a default jump, JMP 010BH takes the program back to the display loop. This jump is only executed if the user presses one of the keys other than 0 , 1 , 2 or NEXT .

If either 0 or 1 were pressed, a jump to 0121H occurred. At that location are the instructions necessary to initialize the computer for one of the most useful subroutines in the monitor, SHIFT AND ENTER. This subroutine is responsible for loading the key closure into the right-most of a set of displays and shifting the former contents of each of the display digits one digit position to the left. The data in the left-most digit is lost. In the form that we use the subroutine, SUB1CH at 807CH, it is necessary to load the H/L registers with the address of the display register that represents the extreme left of the display field. Since our program is written to use the entire set of displays, this will be the left-most digit corresponding to the display register whose address is 000FH. It is also necessary to load the C register with the number of digits in the field. Again, since our program will use all of the displays, this number will be 8. The LXI H, 000FH and MVI C, 08H accomplish these tasks. All that remains is to CALL SUB1CH at 807CH

and the desired loading of the key images into the display registers will be accomplished. A jump back to CONVERT at 010BH completes the display loop. The effect will be that every time the 0 or 1 keys are pressed, that numeral will appear at the right end of the displays, and all of the data already in the displays will be shifted one digit place to the left.

Before leaving the subject of the subroutine SUB1C used for entering and shifting, we should point out that there are two other entrance points that may be more useful for you. These are both used by the monitor as you enter addresses and data. The first is reached by performing a CALL to 8068H. It will enter the contents of the accumulator into the display register corresponding to the right-most display. The field size is only two digits so only two numerals will appear at a time. The initialization of the H/L pair and the C register is done by the subroutine itself; all you supply is the CALL and the data in the accumulator. This is the form of the SHIFT AND ENTER subroutine used by the monitor when you load data into a memory location. The other entrance point is at location 8072H, and like the other, it too sets up the H/L and C registers automatically. This loads the data into a display field four digits wide and beginning at digit no. 6. It is used by the monitor when you load addresses into the computer.

The shifting and entering of 0's and 1's into the displays continues as long as the user wishes. If he makes a mistake, he can simply re-enter the correct number; the incorrect display will be shifted out and lost as the correct displays are shifted in. At some point, the user will wish to set up the carry flag. Although only two of the operations executable from the binary laboratory keyboard make use of the carry flag, the computer does not yet know whether the carry will play a part in those operations or not, so it is ready to accept a carry command. That command takes the form of a depression of the 2 key. When that occurs the instructions already discussed force a jump to 012CH. There the carry will be processed.

Of course, we don't need to actually set the carry flag yet, There will be many instructions before the operations that need the state of the carry are to take place, so just setting the carry would be useless since all those intervening instructions would just change the flag anyway. At this point, all that is necessary is to set the bit in the 001DH memory location reserved for preserving the status of the flags. Since the contents of that location are to be sent to the three LEDs in the left hand bottom corner of the computer, the data contained in that location must be formatted so that it will correctly drive the displays. In particular, the LED assigned to display the state of the carry, the top LED of the three, must receive a 0 to light the LED. The bit that corresponds to this particular LED is the bit no. 3 or the fourth bit from the right. The operation that must result from the depression of the 2 key is the complementing of this bit. Each time we press 2, the condition of that LED should change.

As we have already discussed, the EXCLUSIVE OR function complements selected bits of the accumulator. By loading the accumulator with a LDA 001DH instruction, we have the image of the three LEDs in the accumulator. Since 08H is the hex representation of 0000 1000, the XRI 08H instruction will complement the single bit in the accumulator that corresponds to the CARRY LED. The modified accumulator contents are then reloaded into 001DH and sent to the LEDs at 600H, both with STA instructions. A JMP back to 010BH takes program execution back to the display loop. This means that the CARRY LED can be complemented at any time during the loading of the first binary number into the displays.

When the NEXT key is pressed, a jump to 013AH will occur. This means the user is satisfied with the binary number he has entered into the displays, and the CARRY bit in 001DH has been set according to his wishes. The operations that we plan to experiment with in the binary laboratory all require two operands. The first of these now resides in the displays in the form of 00H's and 01H's. Our first step in processing the operand is to consolidate it in a single 8-bit number in the accumulator. We have faced this problem before and solved it with the subroutine BINARY DISPLAYS TO ACCUMULATOR at 0216H. By performing a CALL to that subroutine, the

displays will be loaded into the accumulator. Since we have need of the accumulator, we will use a MOV B, A instruction to load the operand in the accumulator into the B register where it will stay until we have loaded the second operand. The process of loading the second operand is much the same as the first. The BLANK subroutine is CALLed to erase the prior contents of the display buffers with the seven-segment code equivalents of the display registers, and KEYBOARD-DISPLAY is CALLed to display the contents of the display buffers and wait for a key closure.

The possible key closures are considerably more varied than in the case of the first operand. As before, either the 0 or the 1 key will cause a binary bit to appear in the displays and shift the prior contents one digit to the left. Although the 2 key does not have any significance at this point, keys 9 through F will each cause a different action to take place within the computer. Each of those keys must cause a jump to a different place in the program. With these possibilities in mind we set out to sort out the various key closures.

By performing a CPI 02H followed by a JC 015BH we can use the same trick on the second operand that we did on the first. If either the 0 or the 1 key is pressed, the data in the accumulator will be less than the immediate data associated with the CPI, 02H. In that case the carry flag will be set to 1 and the JC 015BH will cause a jump to 015BH. At that location will be found the same initializing instructions to H/L and C that we used earlier for the SUB 1C or SHIFT AND ENTER subroutine. The subroutine is followed by a JMP back to the display loop. This means that as 0's and 1's are entered from the keyboard they will be shifted into the left end of the displays.

If the JC test checking for 0 or 1 keys fails, program execution will fall through to the next instruction. We now have to determine if the key closure was in the range of 9 to F, inclusive. The same trick can be used again, this time to determine if the key closure was from a key whose value was less than 10H. The CPI 10H and JNC 0144H rules out this possibility by sending pro-

gram control back to the display loop if the key was above the 9 to F range. All that remains is to see if the key closure is a key whose value is larger than 8. So far we have used the carry flag to see if the immediate data was larger than the key closure; now we must do the opposite, that is, use the carry to see if the key closure is larger than the data. Unfortunately, we cannot reverse the process that easily. We have to move the key image out of the accumulator and the D register with a MOV D, A. We can then load the accumulator with 08H with a MVI A instruction. Now when we perform a CMP D, we are in the position of testing the accumulator contents against the key image in the D register. If the key image is larger than 08H, the carry will be set to 1. A JNC 0144H sends the program back to the display loop if the key is 8 or less. Only if the key is in the correct range of 9 to F will 0158H be reached. The jump at this location is to 0166H which is to the keyboard process portion of the program.

There we will perform another CALL to the BINARY DISPLAYS TO ACCUMULATOR subroutine at 0216H. That loads the binary number represented in the displays to be loaded into the accumulator. A MOV C, A loads this number, the second operand, into the C register. We will come back to it in a moment. In the process of checking the key closures we had loaded the key image into the D register. It is still there and in order to determine which key was pressed we're going to have to bring it back to the accumulator with a MOV A, D.

Of course, we could determine the identity of the key whose image is in the accumulator by a set of CPI and JZ intructions, but that would require five bytes for each potential key. Instead we're going to introduce a very powerful concept that will be used in most complex programs. This is a jump table, and it represents a way to go from a byte of data, such as a key closure, to one of many different program segments. That jump table will consist of a single instruction followed by a jump to a common exit point. Each of these program segments will consist of a single byte instruction, like ADD C, followed by a three byte jump to the common exit. Since each of the entries in the jump table is four bytes long, the entry points to the table come in multiples of four bytes. For that reason, the key image of the data in the accumulator has to be modified so

that it matches the four byte increments of the jump table. We need to multiply that image in the accumulator by four so that it matches the table.

In the decimal system, multiplication is generally considered more complex than addition. In the binary system this is not necessarily so. Consider what would happen if we had 01H in the accumulator and wanted to multiply by 2. The binary equivalent of 01H is 0000 0001. Obviously 2 x 1 = 2, and the binary equivalent of 2 is 0000 0010. If you compare the two binary numbers, you should notice that each has a single 1, and in fact, the only difference is in the placement of that 1. If we multiply the result by two again, the result should be four. The binary equivalent of 4 is 0000 0100. Again there is a single 1 and the only difference is in the placement of the 1. As it turns out, to multiply a number by 2 all you have to do is shift it one bit position to the left. In fact, this is the basis for binary multiplication routines. By the same reasoning, division can be accomplished by shifting to the right.

By the example we just went through, we found that to multiply a number by four, all we have to do is shift the number two bit positions to the left. We can accomplish this in the case of the keyboard image by performing two RLC instructions. Since the four left-most bits are 0's they will not affect the result. The two RLC instructions scale the key image correctly; now we have to establish the starting point of the jump table. We do that with an LXI 015FH. Now if you scan the listing of the program you'll find that 015FH is right in the middle of a MVI C instruction and that is certainly not the beginning of the jump table. Would you believe that it really is? The problem is that the jump table in theory starts with key 0 and works up from there. If we had processed a closure of the 0 key, it would have begun at 015FH. The 1 key would have jumped to a spot four bytes later, 0163H, The 2 key would have been four bytes further, 0167H, and so on. By the same reasoning we eventually reach the situation where the 9 key causes a jump to 0183H. While you can do this by adding fours to the hex numbers and working backwards to find the theoretical starting address that must be loaded into the H/L pair, we have

to admit to cheating. A few well placed RST7's, to stop the computer at strategic places, allowed us to arrive at the starting address by experimenting. Not very elegant, but I bet we got there faster than the best pro programmer could have done on paper. Come to think of it, the pro would have done the same thing!

Once the starting address has been loaded into the H/L pair, we add the contents of the accumulator to L. Since the result of the addition is stored in the accumulator, we then have to load this into the L register. At that point the correct jump address is in the H/L register pair. In a moment we will execute a jump to this address in the jump table, which will cause one of seven arithmetic or logical instructions to be executed and the results displayed in the LEDs. Before we can do that, however, we have to get the status of the CARRY LED out of 001DH and set the carry bit accordingly. That way, if the instruction chosen by the keyboard requires the carry flag, it will have been properly set.

The carry flag is set by retrieving the flag status bits from 001DH with a LDA 001DH instruction. The bits in that memory location, other than the carry bit, are zeroed out with an ANI 08H and the zero flag set by ORA A. If the user wishes the carry flag to be set to 1 prior to going into the keyboard-selected operation, he sets the proper bit in 001DH to a 0. The fact that the bit in the memory location is the complement to the desired state of the flag is brought about by the discrete LED requiring a 0 to light up. Hence, a 1 in the carry flag is indicated by a 0 in the memory location, since it is the contents of that memory location that are sent to the LEDs. After the ANI 08H zeroes out everything in the accumulator by the carry bit, we can test it with a JZ 0180H. If we had intended there to be carry bit = 1, the accumulator will now contain 0 (the complementing between flag and LED, remember?), and the JZ will cause the jump to 0180H to occur. There the STC instruction sets the carry flag to 1 and we procede. If the JZ test failed, meaning the user wished a carry = 0, program execution will fall through to 0173H where a 0 will be loaded into the carry flag. In either case, after the carry flag is properly set

according to the user's wishes as expressed by the condition of the discrete LED, the first operand is fetched from its resting place with a MOV A, B and the jump to the table executed with a PCHL. This instruction, you will remember, causes a jump to the address held in the H/L register pair.

We have seen that each of the keys 9 through F will cause a jump to a different place in the jump table. Actually, each of the entries in the table bears a striking similarity to every other entry. Each begins with an arithmetic or logical instruction one byte long followed by a three byte jump to a common exit point, in this case 019FH. The first entry in the table at 0183H is an ANA C instruction. It caused the second operand, contained in the C register, to be ANDed with the first operand, in the accumulator. The result is stored in the accumulator. After the operation is performed, a jump to 019FH occurs. Had the user pressed 9 after loading the two operands, this is the operation that would have resulted.

On the other hand, had the user pressed A , the jump table would have been entered at 0187H which is the location of ORA C; Now the operation will be an ORing of the second operand with the first, and again the results are stored in the accumulator. Each of the keys has a different instruction that will be executed. We have summarized these operations below:

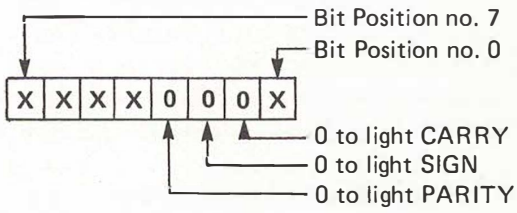
KEY ASSIGNMENT

KEYS		
9	ANA	C
A	ORA	C
B	XRA	C
C	ADD	C
D	ADC	C
E	SUB	C
F	SBB	C

LED ASSIGNMENT

LEDs	
Top	CARRY
Middle	SIGN
Bottom	PARITY

This byte drives the three discrete LEDs.



Don't worry if all these terms are not familiar to you; we'll cover them in agonizing detail in just a second. Under the list of keyboard assignments, we have also indicated the LED assignments. We will cover these in a moment also.

When the operation selected by the keyboard has been performed, a jump to 019FH occurs. At this point we are going to set the three LEDs according to the status of the flags. Since we will need the accumulator for this job, we'll have to save the results of the operation for a moment, a job that is accomplished by MOV C, A. We are now ready to set the three LEDs according to the status of the flags. To turn on the LEDs, it is necessary to send a 0 in the bit position appropriate for the particular LED. We have arbitrarily assigned the three LEDs to the flags as follows. The top LED reflects the status of the carry flag. It is driven by bit no. 3. The middle LED reflects the status of the sign flag. It is driven by bit no. 1. Finally the bottom LEDis assigned the job of displaying the status of a yet to be discussed flag, the parity flag. This is driven by bit no. 2. If, for example, bits 1, 2 and 3 are all 0's, the three LEDs will light. If only bit no. 1 is 0, only the middle LED, assigned to the sign flag, will be lighted.

We will load the data necessary for correctly lighting the LEDs into the accumulator and then send this data to both the LEDs and to the memory location reserved for the image of the LEDs at 001DH. Since the bits that light the LEDs are at the right end of the accumulator, we will set the bits correctly by loading a 0 or a 1 into the carry and then rotating this around to the right end into the accumulator. The first bit to be set is the carry bit. Remember, the CARRY LED is to be lighted if the carry flag is 1. Since a 0 is necessary to light an LED, we need to load a 0 into the carry bit if the carry flag is 1 and a 1 into the carry bit if the carry flag is 0. The first step is to perform a CMC to complement the carry flag. A RAL then shifts this bit into the rightmost bit of the accumulator. Our first flag has been loaded into the accumulator.

The next flag to be considered is the parity flag. This flag is used in certain programs for error checking purposes. One of the most common applications required of a microcomputer is that

of interfacing to other systems. This implies the ability to communicate with those systems, and this has generally been done by some sort of serial data link, either with 20mA current loops such as found on a teletype, or with RS-232C interfaces as found on most modern terminals. In either case, data is transmitted between the microcomputer and the other system, serially, bit by bit. The big worry in this type of serial transmission is dropping a bit because of noise or other external problems. For that reason, many of these serial transmission schemes use data formats whereby start and stop bits begin and end each byte of data and an error bit is included so that a dropped or changed bit can be detected immediately. The error bit is called the parity bit and is a measure of how many 0's or 1's are in the byte. If there are an even number of 1's in the byte, the parity is said to be even and the parity bit is set to 1. If there are an odd number of 1's in the byte, the parity is said to be odd and the parity bit is set to 0. As long as the word consists of an even number of bits, such as a byte which is always made up of eight bits, it makes no difference whether the parity tests the number of 0's or 1's; the resulting parity will be the same.

In practice, the microcomputer assembles the incoming serial bits into a byte, using the start and stop bits to know where one byte begins and the other ends. The parity of the byte thus assembled is then checked against the incoming parity bit. If they are the same, the chances are very good that the byte is error free. It is very unlikely that there could have been two errors in the same byte, and that is the situation that would have to occur to fool the parity checking scheme. In any event, systems used in these applications have to be able to check the parity of the incoming bytes. That could certainly be done with software, but fortunately the CPU used in the ia7301 has some jump, call and return instructions that are conditionally dependent upon the parity of the word in the accumulator. Operating on the contents of the accumulator automatically causes the parity flag to be set according to the number of 0's or 1's in the contents of the accumulator after the operation. Which instructions set or reset the parity flag is detailed on the hex card at the front of the binder.

Now that we have transfered the status of the carry flag into the right-most bit of the accumulator, the carry flag itself can be used for other purposes. In particular, we can use it to transfer the status of the parity flag into the accumulator. To do this we will set the carry to 0 if the parity is even. That is done by the JPE, Jump if Parity Even, instruction at 01A2H. If the result of our arithmetic/logic operation caused the accumulator to have even parity, meaning an even number of 1's or 0's, the jump to 01A9H will occur. At that location is a STC followed by a CMC which results in the carry flag being loaded with a 0. The RAL at 01ABH rotates this bit into the right-most bit of the accumulator and shifts the carry bit already there into the no. 1 bit position. If the parity of our result was odd, the JPE test at 01A2H fails and the program execution falls through to 01A5H where a STC loads the carry flag with 1. A JMP 01ABH then takes the program to the RAL that we wrote into the program to rotate this bit into the accumulator. At the conclusion of this operation, the carry bit will be in the no. 1 bit position of the accumulator and the parity bit will be in the no. 0 bit position.

We still have to load the status of the sign bit into the accumulator. This is done with a JM 01B3H which sets the carry bit to 0 if the sign of the result was minus. Another RAL shifts this bit into the accumulator and moves the carry bit and the parity bit over another bit position in the accumulator.

If the result of our original arithmetic or logical operation was positive, the JM test will fail and the program execution will fall through to 01AFH where a STC sets the carry to 1. A JMP 01B5H takes the computer around the other branch to the RAL which shifts this bit into the accumulator. As a result, the sign bit is shifted into the right-most bit position of the accumulator, 1 if positive and 0 if negative. The three right-most bits of the accumulator are now occupied by the status of the carry, parity and sign flags.

In order to drive the LEDs, the driving bits must be in the no. 3, no, 2 and no. 1 bit positions. At present, the bits are one bit position too far to the right. The RAL at 01B6H corrects this by moving the bits one place to the left. The bits for driving the LEDs are now all in the correct places. However, we know nothing about the state of the other bits in the accumulator. To keep them from affecting the other circuits in the computer we should "disable" them by setting them all to 1's. This is easily done with a ORI F1H instruction. The ORI, OR Immediate, operates by loading the accumulator with a 1 in any bit position previously occupied by a 1 in the accumulator, OR a 1 in the immediate data. The binary equivalent of the immediate data F1H is 1111 0001, so the effect of ORI F1H is to load the right-most bit position with a 1 as well as the four left-most bit positions. The end result, is that all of the bits except the LED driving bits are set to 1. The LED bits have been set to 0 or 1 according to the status of the flags. A STA 6000H sends the data to the LEDs and a STA 001DH updates the image in 001DH.

This takes care of the LEDs reflecting the status of the flags. All that remains is to get the actual result into the display LEDs. A MOV A,C leads our earlier result back into the accumulator and a CALL to DISPLAY BINARY at 0200H causes the result to be displayed in the format on the displays. By then performing a jump back to the display loop in the first operand, we cause the displays to be driven. At the same time the system is now set up so that the user can try the next problem just by keying in data which will form the new, first operand.

The Logical AND. Now that we have explored the workings of the program in detail, we'll use it to learn a little more about the binary number system. The first operation that we want to explore is the logical AND. We have already used this instruction several times in the immediate form, ANI, AND Immediate, where the second operand is supplied as immediate data. Now, we're going to use it in the form ANA C, AND the accumulator with C. This is one of eight ANA instructions ranging from ANA B through all of the working registers including ANA A and ANA M. In each case the contents of the accumulator are ANDed with the contents of the specified

register and the result stored in the accumulator. These instructions set all of the flags according to the result. The ANA A has no effect other than to set the flags; in this respect it is the same as ORA A that we have been using to test the contents of the accumulator. The ANA M instruction ANDs the contents of the memory location specified by the H/L register pair with the contents of the accumulator. As always, the result is stored in the accumulator. Let's try some examples.

Example: Load the first operand with 0000 1111 and the second operand with 1111 0000. Press the key assigned to ANA C. The AND operation results in a 1 in the accumulator only in those bit positions where both operands contain 1's. In this particular example there should be no 1's in the result since no common bit positions loaded with 1's exist.

Enter	See Displayed
CLR EXC	
0 0 0 0 1 1 1 1	

This is the first operand. Now load the second operand.

NXT	
1 1 1 1 0 0 0 0	

Now press 9 to AND the two operands.

9	

Notice that this lights the bottom LED which has been assigned the job of displaying the parity flag. Since the result has an even number of 1's or 0's, in this case eight 0's, the parity is even and the LED verifies this fact.

Example: Load the first operand with 0101 0101 and the second with 1010 1011. In this case, there is only one bit position that has a common 1, the right-most bit. When the ANDing operation takes place, this will be the only bit that remains at 1.

First Operand	0101 0101
Second Operand	1010 1011
ANDed Result	0000 0001

Example: Load the first operand with 0101 0101 and the second operand with 0101 0101. Since these are both the same number, the result should also be the same since each bit position that has a 1 in one operand will also have a 1 in the same bit position of the other operand. When the ANDing operations takes place, every one of these 1's will result in a 1 in the result.

First Operand	0101 0101
Second Operand	0101 0101
ANDed Result	0101 0101

Since there are an even number of 1's in the result, the parity LED lights.

The Logical OR. The logical AND results in a 1 if and only if both operands have a 1 in that bit position. You can remember this by thinking "a 1 in operand A AND a 1 in operand B produce a 1 in the result." The logical OR function is easier to satisfy. It will produce a 1 in the result if

either the first operand OR the second operand has a 1 in a given bit position.

Example: Load the first operand with 0000 1111 and the second operand with 1111 0000. Since there is a 1 in every bit position of either the first or the second operand, there will be a 1 in every bit position of the result. Don't forget to press A instead of 9 for the logical OR operation.

First Operand	0000 1111
Second Operand	1111 0000
OR'd Result	1111 1111

Notice that the even number of 1's in the result causes the parity LED to light. Also, the result is a negative number. You will remeber that, if we choose to interpret a number as having a sign, the left-most bit position, the computer interprets it as a negative number and sets the sign flag which in turn causes the sign LED to light. Whether or not you as a programmer are interpreting the number as having a sign is academic at this point; the computer sets the sign flag regardless. However in practical terms, unless you have included a jump or call dependent upon the sign flag, the computer, and the program, will ignore the sign flag once it has been set. The sign data is there if your program needs to make use of it. If the program does not make use of this data, the flag will be ignored and eventually set to the other state by some other instruction.

Example: Load the first operand with 0101 0101 and the second operand with 0101 0101. What is the result?

First Operand	0101 0101
Second Operand	0101 0101
OR'd Result	_____

Notice that, although the parity LED is lighted because of the even number of 1's in the result, the sign LED is extinguished because the left-most bit position of the result is 0.

Example: Load the first operand with 0000 0000 and the second operand with 1111 1111. What is the result?

First Operand	0000 0000
Second Operand	1111 1111
OR'd Result	_____

Since a 1 in either operand results in a 1 in that bit position of the result, the logical OR can be used to selectively set bits of the accumulator to 1. All that is needed is to make the second operand contain 1's in only those locations that MUST have 1's in the result. The OR function will then modify only those bits of the accumulator selected by the 1's in the second operand. Bits other than those specified by the second operand will be unmodified by the OR regardless of whether or not they are 0 or 1 in the accumulator.

In a sense, this is the exact opposite of the logical AND. Since that operation will set a 1 into the result in the accumulator if, and only if, both first and second operands contain a 1 in that bit position, the logical AND can be used to zero out selected bit positions. All that is required is to specify which bits in the result in the accumulator are to be zeroed out, by loading the second operand with 0's in those positions. Whenever a bit position in the second operand is a 0, the result in the accumulator after the AND operation will also be 0. When the OR or the AND functions are used in this fashion, they are said to be performing a "masking" operation. The first operand in the accumulator is modified by the mask in the second operand. In the case of the OR, the masking operation sets bits to 1's; in the case of the AND, the masking operation sets bits to 0's.

The Logical EXCLUSIVE OR. The final logical operation provided in the computer instruction set is the EXCLUSIVE OR. Whereas the logical OR sets bits to 1, and the logical AND sets bits to 0, the logical EXLUSIVE OR complements bits. Any bit that is 1 in the second operand will cause the corresponding bit in the first operand in the accumulator to be complemented.

Example: Load the first operand with 0000 0000 and the second operand with 0101 0101. What is the result when the EXCLUSIVE OR function is performed? Since the first operand contains all 0's, any complementing action will produce 1's. The second operand controls this function by selecting which bits will be complemented by specifying the bit positions with 1's.

First Operand	0000 0000
Second Operand	0101 0101
EXCLUSIVE OR'd Result	0101 0101

The parity LED is lighted because the result contains an even number of 1's.

Example: Load the first operand with 1111 1111 and the second operand with 0101 0101. Now the fact that the first operand is all 1's, means any complementing action will produce a 0. Since that complementing action is specified by the 1's in the second operand, the result will be mixed 0's and 1's.

First Operand	1111 1111
Second Operand	0101 0101
EXCLUSIVE OR'd Result	1010 1010

Since the result contains an even number of 1's, the parity LED is lighted. Also, the fact that the left-most bit position of the result is a 1 causes the sign LED to light indicating that the result is negative.

Example: Load the first operand with 0101 0101 and the second operand with 1010 1010. What is the result?

First Operand	0101 0101
Second Operand	1010 1010
EXCLUSIVE OR'd Result	_____

Example: Load the first operand with 0101 0101 and the second operand with 0101 0101. What is the result?

First Operand	0101 0101
Second Operand	0101 0101
EXCLUSIVE OR'd Result	_____

The last example is important because it proves that any number EXCLUSIVE OR'd with itself is zero. This is an easy way to zero out the accumulator. A MVI A, 00H requires two bytes in memory, but a XRA A is a single byte instruction.

ADDITION IN BINARY. That completes the discussion of the logical operations in the computer and brings us to the very important subject of arithmetic. Here is the stuff of which calculations are made. Somewhere in most of our programs will be arithmetic operations, sometimes very long and complex. But all of those arithmetic operations are done in the binary system and converted at some point to decimal. The most basic of the arithmetic operations is addition, and in the ia7301 this comes in two varieties, with and without the carry.

When you add two numbers together you sum the individual digits. Sometimes a carry in one of the columns is generated and has to be added to the next column. The two instructions in this computer are ADD and ADC, ADD with Carry. Together they allow you to perform addition of large multibyte numbers. Let's cover ADD first. This instruction adds the first operand to the second and stores the result in the accumulator. As with the logical instructions, there are eight ADD instructions that allow any of the registers or memory locations, in the case of ADD M, to be added to the first operand which is always in the accumulator. Another variety is ADI which supplies the second operand in the form of immediate data. If you perform ADD A, the contents of the accumulator become both the first and the second operand. The effect is to double the previous contents of the accumulator.

Although you may not realize it, you already know how to add binary numbers. The binary addition table is the same as the decimal addition table, just much shorter. What do you get if you add 0 and 1? 1 of course. What about 0 and 0? Again the answer is easy. See, binary addition is the same as decimal. Oh, just one thing. When you add 1 and 1 in decimal you get 2, but as you know there is no 2 in binary. But the binary equivalent of 2 is 10. You can think of this as 1 + 1 = 0 carry 1. Let's try some examples.

Example: Add 0000 0001 in the first operand to 0000 0000 in the second operand. Don't forget to press C to perform the ADD instruction.

First Operand	0000 0001
Second Operand	0000 0000
ADDed Result	0000 0001

This proves our earlier supposition that 0 + 1 = 1.

Example: Add 0000 0001 in the first operand to 0000 0001 in the second operand.

First Operand	0000 0001
Second Operand	0000 0001
ADDed Result	0000 0010

There are two ways you can think of this. 1 + 1 = 2 and the binary equivalent of 2 is 10. Or, alternately, 1 + 1 = 0 carry 1. Both lead to the same result. Now let's add 1 to the result of this example.

Example: Add 0000 0010 in the first operand to 0000 0001 in the second operand. This is the equivalent of adding 1 to 2, and we already know the answer to that. We should also be able to predict the contents of the displays since we know from our earlier exercises what the binary equivalent of 3 is.

First Operand	0000 0010
Second Operand	0000 0001
ADDed Result	0000 0011

Notice that we can solve this problem on paper by adding the individual columns. The right-most column is 0 + 1 = 1 and the next column is 1 + 0 = 1.

Example: Let's try one more in this series. We already have the binary equivalent of 3 in the first operand. Let's add 1 to it and see if we can predict the result. Load 0000 0011 into the second. What is the result of ADDing the two?

First Operand	0000 0011
Second Operand	0000 0001
ADDed Result	0000 0100

If we perform the operation column by column, the first column is 1 + 1 = 0 carry 1. The second column must now have the carry from the result of the first added into it. When we include the carry with the two operands, we get 1 + 1 + 0 = 0 carry 1. The third column is now 1 + 0 + 0 = 1. The "pencil and paper" result matches the one we found when we used the computer to try the problem.

Example: This brings up an interesting point we have not had time to consider before. If the carry from the previous column has to be added in, what would happen if both operands had 1 in them. The column would then become 1 + 1 + 1 =? Actually you already know the answer to this one. We know that 1 + 1 = 10 since we tried that in our second ADD example. The question is then what is 1 + 10 = ? The third example provides the answer to that, 1 + 10 = 11. You can think of 1 + 1 + 1 as 1 + 1 + 1 = 1 carry 1. To prove it, use the binary laboratory program. Load the first operand with 0000 0011 and the second operand with 0000 0011. What is the result when we ADD them?

First Operand	0000 0011
Second Operand	0000 0011
ADDed Result	0000 0110

The first column is 1 + 1 = carry 1. The second column is an example of the situation we just covered, 1 + 1 + 1 = 1 carry 1. The final column then becomes 1 + 0 + 0 = 1.

Example: Load the first operand with 0000 0001 and the second operand with 1111 1111. What is the result if we ADD the two together? This is a problem similar to adding 1 to 9999 9999 in decimal. Each column generates a carry in that column too. This has to end somewhere and it does. We add one more column and place a 1 in it, so that in decimal the solution is 1 0000 0000. But we cannot really add a column to the accumulator, or for that matter any other registers in the CPU either. The best that we can do is to set the carry flag to indicate an overflow occured.

First Operand	0000 0001
Second Operand	1111 1111
ANDed Result	0000 0000

As expected, the CARRY LED is ON indicating an overflow out of the accumulator. The PARITY LED is also ON since there are an even number of 0's or 1's (in this case, eight 0's).

Example: Load the first operand with 1010 1010 and the second operand with 1010 1010. What is the result when we ADD them together?

First Operand	1010 1010
Second Operand	1010 1010
ADDed Result	_____

THE ADD WITH CARRY, ADC. It is easy to see how the ADDition of larger numbers that use the full capacity of the accumulator can require the carry flag to be set. If the number is very large, say 24 bits long, we could perform the addition one byte at a time. First the two least significant bytes of the operand would be added. Then any carry from the addition of the first bytes would be added to the second byte of one of the operands, the second bytes then added

themselves, the carry from the second bytes added to the third bytes, and so on. It is easy to see how the addition of multibyte numbers can be handled.

Fortunately the instruction set of the computer makes this process even easier by including an instruction, ADC, ADD with Carry, that includes the carry flag in the addition. To add a pair of multibyte numbers together, we now add the two least significant bytes of the operands with an ADD instruction. Since this instruction ignores the status of the carry flag going into the addition, we will not have to set the carry to 0 before beginning the problem. ADD does set the carry according to whether or not the result overflows the accumulator, however, so this data has been loaded into the addition of the second bytes. For this addition we use the ADC instruction. It adds the carry and the two operands all at once and sets the carry according to the result. Let's see how it works.

Example: Load the first operand with 0000 0000 and set the carry flag to 1 by using the 2 key. This should light the CARRY LED. Load the second operand with 0000 0000. What is the result of ADCing the two operands and carry?

First Operand	0000 0000
Second Operand	0000 0000
Carry	1
ADCed Results	0000 0001

Obviously the ADC instruction adds the prior contents of the carry flag with the operands when it performs the addition. Since both operands in this case were zero and only the carry flag was 1, the results were 0000 0001. Since no carry out was produced by the operation, the carry flag is reset to 0 and the LED turned OFF. You can try the same operation, if you wish, using ADD instead of ADC. In the case of ADD the prior contents of the carry flag are ignored and the result will be 0000 0000.

Example: Load the first operand with 0000 0001 and set the carry flag to 1. Load the second operand with 0000 0001. What is the result of ADCing the two operands and carry?

First Operand	0000 0001
Second Operand	0000 0001
Carry	1
ADCed Results	0000 0011

This is another example of the 1 + 1 + 1 = 11 problem that we discussed earlier. The ADC instruction produces consistent results.

Example: Load the first operand with 1111 1111, the carry with 1 and the second operand with 0000 0001. What is the result of ADCing the two operands and carry?

First Operand	1111 1111
Second Operand	0000 0001
Carry	1
ADCed Results	0000 0001

The carry flag is set as a result of the overflow out of the accumulator.

The only difference in the ADD and the ADC instructions is in the way they treat the carry flag before the addition. The ADD instruction ignores it; the ADC adds it in as though it were a third operand. Both set the carry according to the result.

BINARY SUBTRACTION. We come now to one of the most confusing aspects of binary arithmetic, subtraction. We will assume that you can figure out the basic rules of binary subtraction

from the addition problems that we just worked, for instance, 1 – 1 = 0 and so on. Instead of spending a lot of time developing those rules what we will do is to develop a method of performing 8-bit, and even larger, subtractions.

The subtraction methods that we discuss use the complementing process. In these operations it will be necessary to complement the entire contents of the accumulator. The ia7301 has a special instruction just for this purpose, CMA, Complement Accumulator. When it is executed every 1 in the accumulator will be changed to a 0 and every 0 to a 1. None of the flags are affected by the CMA operation, so that any data in the carry flag will be there after the operation as well. For instance, if the accumulator contained 0000 1111 and a CMA was performed, it would then contain 1111 0000. This complement is known as the 1's complement. Thus 1111 0000 is the 1's complement of 0000 1111.

The system of subtraction that we will use is based on the 2's complement. This is performed by adding 1 to the 1's complement. Thus the 2's complement of 0000 1111 is 1111 0001.

Example: What is the 2's complement of 1010 1010?

Operand	1010 1010
1's Complement	0101 0101
Add 1 to form 2's Complement	1
2's Complement	0101 0110

Example: What is the 2's complement of 0110 1111?

Operand	0110 1111
1's Complement	1001 0000
Add 1 to form 2's Complement	1
2's Complement	1001 0001

We could subtract two numbers by using the rules of subtraction, but computers seldom have special subtract circuits built into them. Instead they perform subtraction by adding the 2's complement of the second operand to the first. When this done there will frequently be an overflow carry generated by the addition. The presence of a carry, that is the carry bit set to 1, indicates NO BORROW. The opposite state, the carry bit set to 0, indicates a BORROW has occurred. This is somewhat confusing since it is the opposite of the addition process. Just remember, in subtraction the carry bit indicates a borrow if 0 and no borrow if 1.

The process is more confusing yet. If no borrow has occured, the answer will be the 2's complement of the answer and must be converted. Let's try a few examples to see if we can make this clearer.

Example: Subtract 3 from 5. The binary equivalent of 5 is 0000 0101. The equivalent of 3 is 0000 0011 and the 2's complement of this is 1111 1101. Adding the two . . .

First Operand (5)	0000 0101
Second Operand (2's of 3)	1111 1101
	1 0000 0010

The result is correct. 0000 0010 is the binary equivalent of 2 and the carry out of the addition indicates that no borrow occurred and the answer is correct as it stands.

Example: Subtract 1 from 8. The binary equivalent of 8 is 0000 1000. The equivalent of 1 is 0000 0001 and the 2's complement of this is 1111 1111. Adding the two . . .

First Operand (8)	0000 1000
Second Operand (2's of 1)	1111 1111
	1 0000 0111

Again the result is correct. 0000 0111 is the binary equivalent of 7 and the carry out of the addition indicates that no borrow occurred and the answer is correct as it stands.

Example: Subtract 0 from 5. The binary equivalent of 5 is 0000 0101. The equivalent of 0 is 0000 0000 and the 2's comlement of this is 1 0000 0000 where 1 is in the carry bit. Adding the two . . .

First Operand (5)	0000 0101
Second Operand (2's of 0)	1 0000 0000
	1 0000 0101

The result is the binary equivalent of 5. The carry bit is properly set to 1 indicating no borrow occurred.

Example: Subtract 8 from 5. This will generate a negative answer, minus three. The binary equivalent of 5 is 0000 0101. The binary equivalent of 8 is 0000 1000 and the 2's complement of this is 1111 1000. Adding the two . . .

First Operand (5)	0000 0101
Second Operand (2's of 8)	1111 1000
	0 1111 1101

This certainly doesn't look like –3. Ah, but the 0 in the carry bit indicates a borrow has occurred so we must convert the answer back to its 2's complement.

Operand	0 1111 1101
1's Complement	0000 0010
2's Complement	0 0000 0011

This corrects the answer. 0000 0011 is the binary equivalent of 3 and the 0 in the carry bit indicates a borrow had occured, as indeed it would have, since 5 – 8 is a negative number.

This is the basis for subtraction in the computer. Fortunately we don't have to go through this when we subtract numbers in the ia7301 since it has subtraction instructions in its instruction set. But those instructions execute by forming the 2's complement of the second operand and adding it to the first. The understanding that we have gained in this form of arithmetic will stand us in good stead when we begin to subtract and work with larger numbers.

There is one point of potential confusion that must be cleared up before we go on. While the addition operation sets the carry bit to 1 whenever there is an overflow from the accumulator, the subtraction operation does quite the opposite. When an overflow occurs in subtraction, we have already seen that this means no borrow has taken place. However, the CPU sets the carry flag to 0. The carry flag then becomes an indicator of borrow in the following sense, a 1 in the carry flag means a borrow has taken place, a 0 in the carry flag means no borrow has taken place. The easiest way to understand this is to think of the carry bit and the carry flag as two separate entities. Always before, and particularly in addition, the two are identical. If an overflow from the accumulator loads a 1 into the carry bit, the computer has set the carry flag to 1 also. However in subtraction the opposite is true. When an overflow from the accumulator sets the carry bit to 1, meaning no borrow, the computer sets the carry flag to 0.

Example: Subtract 3 from 5. Load the first operand with the binary equivalent of 5, 0000 0101, and the second operand with the binary equivalent of 3, 0000 0011. This is the same as the first pencil and paper subtraction example we did a few pages ago. Execute the subtraction by pressing E.

First Operand (5)	0000 0101
Second Operand (3)	0000 0011
SUBed Result	0000 0010

This is certainly the binary equivalent of 2 as we had expected. We know from the earlier example that the 2's complement subtraction set the carry bit to 1 but now, since this is a subtraction operation, the computer sets the carry flag to 0 even though the carry bit is 1. Thus the fact that the CARRY LED is not lighted means the carry flag is 0 and no borrow has occured.

Example: Subtract 1 from 8. Load the first operand with the binary equivalent of 8, 0000 1000, and the second operand with the binary equivalent of 1, 0000 0001. This is the same as the second example we did earlier with paper and pencil.

First Operand (8)	0000 1000
Second Operand (1)	0000 0001
SUBed result	0000 0111

As we expected, this is the binary equivalent of 7. Since no borrow occurred the carry flag is 0.

Example: Subtract 0 from 5.

First Operand (5)	0000 0101
Second Operand (0)	0000 0000
SUBed Result	0000 0101

Example: Subtract 8 from 5.

First Operand (5)	0000 0101
Second Operand (8)	0000 1000
SUBed Result	1111 1101

This time the CARRY LED is lighted indicating a borrow has occurred, and we have to convert the result in the accumulator into its 2's complement. The easiest way to do this is with a CMA instruction followed by INR A. When you do that you'll find 0000 0011 in the accumulator which is the binary equivalent of 3. The carry flag is still set to 1 indicating a borrow occurred and we should consider the converted result as a negative number.

MULTIBYTE SUBTRACTION. What happens when we want to subtract larger numbers, say several bytes long? The computer has an instruction for that, too. It's called SBB, Subtract with Borrow, and it can make the whole process a lot easier in the same way the ADC instruction helped with multibyte addition. Whenever multibyte numbers are subtracted, the process always begins with the least significant bytes. A SUB instruction is perfect for that, since we don't need to worry about the state of the carry flag going into the problem. If no borrow occurs during that subtraction of the first byte, we can go on to the second. If a borrow occurs, indicated by the carry flag being set to 1, we have to correct the result of that operation by forming the 2's complement of that byte. We can then go onto the second byte, knowing that, borrow or not, the first byte result is now correct. For the second byte we'll use a SBB instruction, so that the effect of the first byte borrow will be included in the calculation. The SBB instruction operates by subtracting the carry flag from the first operand along with the second operand. This adjusts the second byte for a borrow that may have come from the result of the first byte operation. If a borrow occurs as a result of the second byte subtraction, the third byte will have to be adjusted in the same way.

Example: Subtract 5 from 8. Assume that this is a second byte operation and that a borrow in the first byte had occurred. This might have happened, for instance, if the entire problem was 86 – 59. When 9 was subtracted from 6 the carry flag would have been set to 1 because of the borrow. Now that borrow has to be included in the subtraction of the second byte. Load the first operand with 0000 1000, the binary equivalent of 8. Set the carry flag as though the borrow had occurred by using the 2 key. Then load the second operand with 0000 0101, the binary equivalent of 5. Perform the SBB operation by pressing the F key.

First Operand (8)	0000 1000
Second Operand (5)	0000 0101
SBBed Result	0000 0010

This is the binary equivalent of 2, and if you had mentally performed the 86 – 59 subtraction, you would agree that 2 is the answer we were looking for. Since there was no borrow in this operation, the carry flag is reset to 0 and the CARRY LED turned OFF.

DEBUGGING PROGRAMS. We've written and worked on quite a few programs now, and you have probably tried a few of your own. If you have, you've also discovered how difficult it can be to find an error in a program and correct it. The process of tracking down those inevitable errors and correcting them is called "debugging," It is usually the longest phase of the program writing cycle.

Occasionally one runs into an instruction that should not be in the program. The easiest thing to do at that point is replace it with a NOP and go on. More often you'll find a program segment that has an error. If the correct version is shorter it can be loaded in place of the incorrect version, and the unused locations replaced with NOPs so the program will execute correctly.

Unfortunately, the correct versions generally turn out to be longer than the original. When the offending program segment happens to be the last segment of the program, the fix is easy. Just correct the program segment. Since there is nothing after a segment, there is no reason why the program can't run a little longer.

If, as usually happens, the offending segment is right in the middle of the program, it cannot be simply stretched out to include the corrections. Of course, we could rekey in the entire program after that point to make room for the corrections, but this is very time consuming. And anyway, five minutes after you do key in the program again, you'll find another bug and have to repeat the whole process again. There is a way out, but it requires discipline on your part. All you have to do is load in the corrected program segment where the old one was. As you start to run out of room, since the corrected version is longer than the original, load a JMP to an unused portion of memory. Continue the program segment in the new memory area. Of course, you'll have to provide a way back to the original program. If the error was in a subroutine, the normal RET that concludes the subroutine will provide the exit to the correct place in the CALLing program. If the error was in the middle of a segment, another JMP at the end of the corrected version in the new memory area will take the program execution back to the correct point in the origianl area. The discipline comes in because you have to document this in your notes very carefully. You probably won't want to leave all of these "patches" in your program, but they do allow you to fix a half-dozen or so bugs before keying in the entire program again.

BREAKPOINTS. We have already learned that we can make our program stop at the point we wish by inserting a RST 7, which has the code of FFH, into our program. This is called a "breakpoint," since it breaks the program and allows us to examine it at that point. RST 7 is one of a class of eight RST instructions. They all operate be causing the computer to save the contents of the program counter in the stack memory, and then jump to a point in the early RAM memory locations. In the case of RST 7, this point is 0038H. It is up to the programmer to put a meaningful program segment at that point. In the case of the ia7301, location 0038H is in RAM

memory and any instructions there will be lost when the power to the computer is turned off. For that reason, as part of the start-up procedure the monitor program loads a JMP instruction at 0038H. The jump is to a point in the monitor that causes the single-step function of the computer to operate.

But what if we didn't have that step routine built into the monitor? Could we write one and use it to debug programs? Sure, and it's not as hard as you might think! Consider what we want that routine to do. We will load it at 0038H where it'll be out of the way of the regular program, but will be automatically entered every time the computer finds an FFH in the user programs. That subroutine must save the values of the CPU registers, including the stack pointer and the program counter, in the memory. In the case of the step function in the monitor, these memory locations are located right over DIRG7, from locations 0010H to 001BH. When you press the DCR key, the monitor loads the contents of these locations into the displays, one at a time.

One of the first steps will be to load the H and L registers into memory. This is a little trickier than it looks. We have normally moved registers to memory with a MOV M, R instruction, but that requires that the H and L registers be pointed to the correct location in memory, and of course, that would destroy the prior contents of the accumulator.

Fortunately there is an instruction, SHLD, Store H and L Direct, that solves the problem neatly. SHLD is a direct instruction meaning the address must be supplied with it in the form of immediate data. That allows us to write SHLD 0050H and have the H and L registers saved at 0051H and 0050H respectively. The process leaves the rest of the registers and the flags undisturbed, but frees up H and L. That means we can load them with the addresses of the memory locations reserved for the other registers, and then load the registers into these locations with our familiar MOV M, R instructions. Those locations are undoubtedly adjacent, so we can increment H/L to

point at the next location with an INX H instruction which does not disturb the flags. In this way we can load B, C, D and E into memory. To get the flags into the memory is a little trickier. Although there are no direct instructions for loading the flags into a specific memory location, or for that matter, into a specified register, we can load the accumulator and the flags together into the stack memory with a PUSH PSW instruction. If this is followed with a POP D, the effect will be to load the accumulator into one of the registers and the flags into the other. Once in the D and E registers, they can be loaded into the memory locations by additional MOV M, R instructions.

That takes care of the CPU registers, but what about the program counter and the stack pointer? As it turns out, these are considerably easier than you might think. When the FFH was encountered, the computer acted as though it were executing a CALL to 0038H. In so doing, it stored the correct return address, the contents of the program counter, in the stack memory. That return address is at the top of the stack and if we now execute a POP H instruction, the return address will be loaded into H/L. Another SHLD will load this address, the original contents of the program counter, into the correct memory location reserved for it. So much for the program counter.

To load the value of the stack pointer into the memory, we first have to gain access to it. Although there are no instructions for directly loading the stack pointer into the other working registers where we can move it to memory, the DAD SP instruction has possibilities. This instruction adds the 16-bit contents of the stack pointer to the H/L register pair and saves the results in H/L. If H/L were to be zeroed out first, DAD SP would in effect move the contents of the stack pointer into H/L. The zeroing operation is done simply by an LXI H, 0000H. When the DAD SP is executed, the stack pointer will be loaded into H/L, and a pair of MOV M, R instructions will then load the stack pointer into the memory registers reserved for it.

Now that all of the registers are saved in the memory, they can be examined and changed by very much the same mechanism used by the DCM mode. After all, the memory locations hold images of the registers, certainly not the real registers, and the computer cannot, and need not, distinguish between those images and normal data. It cannot change them at will.

Of course, at some point we'll have to go back to the program and continue execution. To do that, we need to load the registers with the images of the registers from the memory locations. This is largely a matter of repeating the process we just went through in reverse. One exception is the DAD SP that we used to load the stack pointer into the H/L registers which is not reversible. We'll have to find another way of loading the contents of the H/L registers into the stack pointer. As you might have expected, there is an instruction just for that purpose. SPHL loads the stack pointer with the contents of H and L. All we have to do is execute a LHLD, Load H and L Direct, the instruction complement of the SHLD. This loads H/L with the contents of the memory locations specified by the direct data accompanying LHLD. The SPHL then loads the contents of H/L, just fetched from the memory locations holding the image of the stack pointer, into the stack pointer itself. Another LHLD can be used to load the H/L registers with the image of the program counter from the memory locations reserved for it. ~~This is PUSHed onto the stack pointer itself. Another LHLD can be used to load the H/L registers with the image of the program counter from the memory locations reserved for it.~~ This is PUSHed onto the stack with a PUSH H instruction. This process has the effect of getting the former address in the program counter back to the top of the stack where it can be treated like a return address. Once the stack pointer and the program counter have been set up, the H/L register pair is free for bringing the data into the other registers with MOV R, M instructions that are the reverse of the process used to save those registers. A final LHLD loads the H/L registers themselves. All that remains is to execute a RET to balance the stack from the earlier PUSH H and to load the return address into the program counter.

SINGLE-STEP. The subroutines just described are basically the ones in the monitor program that service the STEP key. We have discussed the first phase of loading the register contents into memory locations as a reaction to the computer finding a FFH or RST7 in the program. The assumption is made that the programmer has placed that RST7 there as a breakpoint so that the program will cease execution at that point so it may be examined and debugged.

There is a second way to get the RST7 into the program besides loading it there during debug with the DCM mode. The STEP key on the computer electronically jams an FFH onto the CPU overriding whatever data or instruction were being fetched from memory at that time. The effect on the computer will be the same as if the RST7 had been in the program all along. The effect to the programmer, however, is that he can stop the computer at will by pressing STEP. Both methods have their place. The STEP key is perfect for tracing a program segment, instruction by instruction. Loading the RST7 into the program with the DCM key is handy for stopping the computer at some point well within the program. Then it can be single-stepped to find the bug.

INTERRUPTS. The process by which the STEP key jams the RST7 into the computer is called "interrupting" the computer. This is a very powerful technique which allows the computer to be doing one job, such as keeping time on the displays, and then when an interrrupt signal is received, dropping its first task and executing a second task. When the second task has been completed, the computer returns to the first. In this mode, an interrupt is treated exactly as though it were a subroutine CALL. The return address is saved on the stack, so that the service subroutine written to service the interrupt should be terminated with a RET instruction to get program execution back to the real-time program. It will undoubtedly be necessary to save all the registers as part of the service subroutine, since you have no way of knowing where in the real-time program the interrrupt will occur. This means that you have to assume the worst case and save all the registers since they may be in use in the real-time program.

One of the most common uses for the interrupt capability is as a "handshaking" line. When data is being read from an input port, the computer has no way of knowing when the data is complete and ready to read. After all, when the peripheral equipment supplying the data loads its output bits into its interface circuits, some bits may be there before others, which could cause errors. Then the data must remain stable until it has been read. If the computer is programmed to read the input port continuouly, it will reread old data for some time, then catch data in transition, which may or may not be correct, and finally read the new data. With the interrupt system, the peripheral supplying the data waits until the data on the input lines is correct and ready to read. It then sends an interrupt signal to the computer which drops what it is doing, reads the data once, processes it and then returns to its original task. When the peripheral has another byte of data ready, it interrupts the computer again, etc. In this way the interrupt signal is used as a "handshaking" signal by the peripheral to tell the computer when to read the input port.

Although your ia7301 was not designed with handshaking ports, you can add this to your system very simply by adding one wire to the PC board. This wire has the same effect as the STEP key and you use it as a handshaking line for the I/O ports of the computer. Every time that line is pulled to a logic 0 level, (less than 0.4V), an interrupt will occur and the computer will jump to 0038H to service the interrupt. At that location the monitor has written a 3-byte jump instruction which takes program execution to the single-step routine in the monitor. You'll have to load your service routine right over that jump at 0038H because you want to treat the interrupt a lot differently than the monitor does. Once that has been done, you cannot press the CLEAR key since that will go back and write the jump to monitor right back at 0038H again. If you need to set the program counter and the stack pointer back to 0100H, you'll have to use the DCR key to do it. The hookup point for the interrupt wire is shown if Fig. 9-6.

THE INTERRUPT INSTRUCTIONS. The interrupt systems are so important in the computer that there have been some special instructions added to the instruction set just for processing

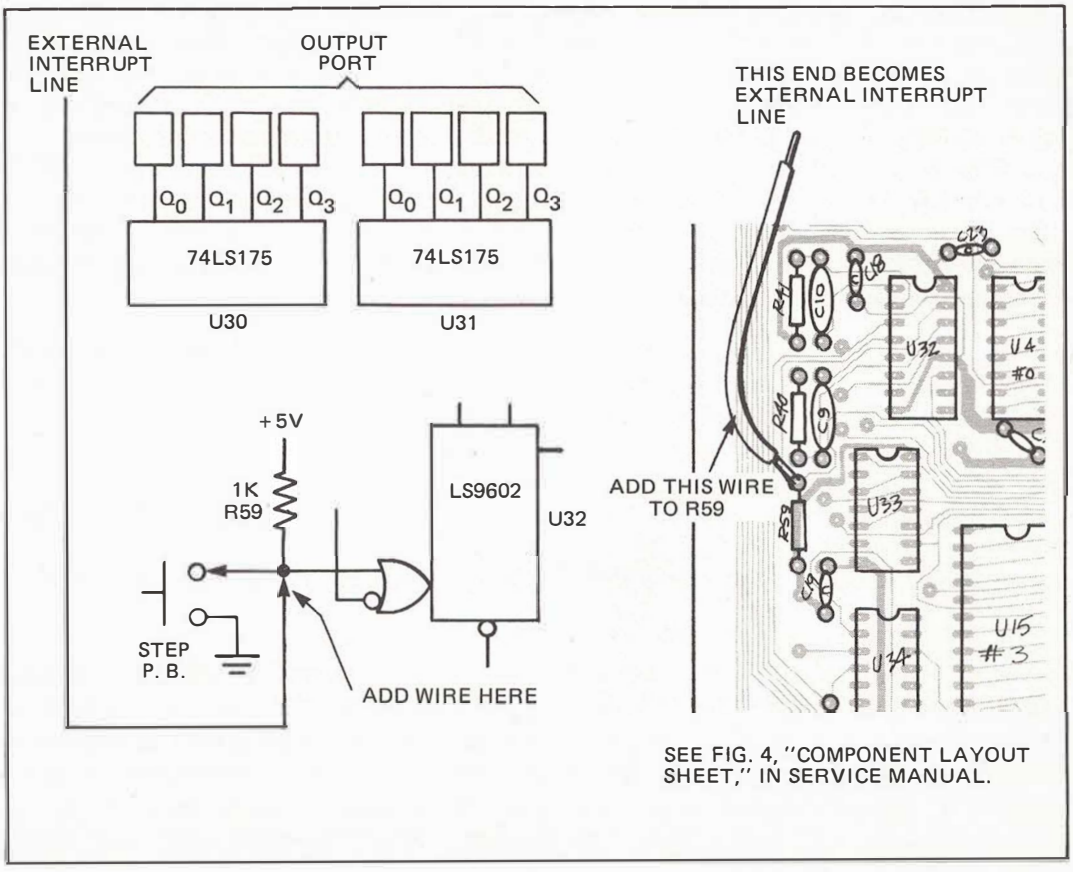


Fig. 9-6 Schematic and pictorial showing how to add an external interrupt line to the ia7301.

them. To understand the function of these instructions, we need to understand a little more about how the interrupt process works. We know that when an interrupt signal is received the CPU completes the instruction it is executing at that time and then jumps to a point called the "restart vector". In the case of the RST7, which is the interrupt that the ia7301 responds to, the restart vector is 0038H. At that point will be the beginning of the interrupt service routine. The RST7 does not need this address because it always causes a CALL to the same point, 0038H. Because RST7 is a CALL, the return address is saved in the stack memory so that the RET that terminates the interrupt service subroutine can get the program back to the main program. A normal subroutine begins with a series of PUSH instructions that save the registers in the stack memory so they can be used by the subroutine.

What would happen if another interrupt occurred during the process of PUSHing the registers onto the stack? The computer would go back to 0038H and the registers would have only been half saved. The stack would be unbalanced and the program would be hopelessly fouled up. This problem never arises in a program without interrupts because program execution always follows a predetermined path. But the interrupt may occur at any time; It is said to be "asynchronous". To prevent an unwanted interrupt from occurring just as we are saving the registers from another interrupt, the CPU has the ability to lock out interrupts with a special latch inside the CPU. This is the interrupt enable flip-flop, and while the flip-flop is disabled, it will not respond to any interrupts.

The interrupt enable flip-flop is set and reset with a pair of instructions, EI, Enable Interrupts, and DI, Disable Interrupts. When the computer is first turned on, the interrupt flip-flop will automatically be disabled by the CPU. After the initializing part of the program has taken place, an EI instruction sets the computer up so that it can accept and process interrupts. If certain program segments or subroutines cannot be interrupted for whatever reason, the interrupts can be locked out by executing a DI instruction. This disabled state will continue until an EI instruction is executed.

When an interrupt occurs, the enable flip-flop is automatically disabled. This prevents further interrupts from coming along in the middle of the register saving process. While this automatic feature of the CPU is usually convenient, you must remember to always include an EI instruction somewhere within the interrupt service subroutine, or the system will never respond to another interrupt.

Finally, the situation occasionally arises where we do not want the microcomputer to begin executing its program when power is first applied. In these types of systems, a RESET button is usually supplied as part of the front panel, and program execution does not begin until that button is pressed, regardless of when power was first applied. One way to accomplish this is to continuously read the RESET button, looping until a closure is found. Then and only then will the computer begin the real program. Another way is to execute a HLT, Halt, instruction. This causes the computer to stop and enter a WAIT state until an interrupt is received. At that point the computer returns to its normal stae and begins to execute instructions again. If the RESET key causes an interrupt, this will automatically start the computer again. Used in this fashion, the HLT operates by stopping the computer until an interrupt, such as a prompt from the user occurs. Since only an interrupt can cause the computer to exit from this state, it is crucial that the interrupts be enabled with an EI instruction prior to the HLT instruction.

PRIORITY INTERRUPTS. Some very large systems will have more than one interrupt. These interrupt lines, usually eight in number, are all assigned priorities by the circuits that make up the computer. By carefully using low priority interrupt lines with low priority ports, the system can handle many tasks of varying priority. A high priority interrupt can interrupt the service subroutine of a low priority interrupt, but not vice versa.

Each of the interrupt lines is assigned an interrupt vector of its own. These occur in low order memory locations, and each provides a jumping off point to the service subroutine for that particular interrupt line. Each interrupt causes a different code to be jammed into the computer. As we have already seen, the interrupt associated with RST7 causes the computer to execute a CALL to 0038H. Another interrupt line causes an F7H to be jammed into the computer. This is the code for RST6 and it causes the computer to execute a CALL to a different location somewhere between 0008H and 0038H. While these are included in the instruction set because they are recognized and executed as such by the computer, they are not really intended to be written into programs by the programmer. Instead, they are codes that are to be jammed into the computer by the circuits associated with the interrupt lines. Because the ia7301 uses a RST7 to enter the single-step mode, this particular instruction is useful either as a code jammed into the computer by the STEP key or as an instruction the programmer can treat as a breakpoint.

Priority interrupts belong to large complex microcomputer systems. In most systems, like the ia7301, there is only one level of interrupt, usually the one associated with RST7. This does not mean that there can be only one interrupting source, It does mean, however, that all interrupts are processed from a common entry point. The beginning of the interrupt service subroutine must "poll" the various interrupting sources to see which causes the interrupt. This method is considerably cheaper than the priority system. although the software associated with the polling technique makes it slower than the priority system. For some types of systems, where communication with a high speed peripheral is necessary, the time required to poll the interrupting sources may make the technique too slow to be practical. This is a complex matter and is usually addressed by the hardware designer and the software programmer working together.